Computational Storage API

Version 1.0.10

ABSTRACT: This SNIA Standard defines the interface between an application and a Computational Storage Device (CSx). For each CSx there needs to be a library that performs the mapping from the functions in this standard and the CSx on the specific interface for that CSx.

Working Draft

January 28, 2025

DISCLAIMER

The information contained in this publication is subject to change without notice. SNIA makes no warranty of any kind with regard to this specification, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. SNIA shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this specification.

Table of Contents

Table of Figures

1 Scope

This document describes a software Application Programming Interface (API) for a Computational Storage Device (CSx). This is the base set of functions and additional libraries are able to be built on this set of functions.

Familiarity with storage and filesystems usage is desired. An understanding of how compute and memory may be utilized in an application and a sound understanding of the Operating System environment is required. Applications of computational storage, although not typically restricted, apply to Enterprise and Datacenter usages and applications in high-performance and datacenter environments.

This document is intended for members of the SNIA workgroup and its associates.

1.1 Obsoleted functions

The following functions are obsoleted in this version of this standard. The definitions of these functions are specified in a previous version of this standard.

Table 1: Obsoleted Functions		
Function name	Last version where specified	Replaced by
csAddBatchEntry()	Version 1.0	csConfigureBatchEntry()
csHelperReconfigureBatchEntry()	Version 1.0	csConfigureBatchEntry()

1.2 About the Computational Storage API

The Computational Storage (CS) API defines a set of functions that provide a standardized way to access compute offload capable devices. The CS API standard is based on the SNIA Computational Storage Architecture and Programming Model. CSxes may be directly attached, network attached, or fabric attached. This standard applies to CSxes using any type of connection with the aim of providing an interface that is seamless while standardized across all such CSxes.

Additionally, the CS API may provide an interface that is able to also work when the application is in transition and does not have a device-based offload mechanism in place. For such cases, a host CPU based mechanism may be substituted for a device-based implementation without changing the interface.

1.3 Document layout

This document is organized to provide familiarity with device types, function usages, function definitions and sample code.

2 Definitions, abbreviations, and conventions

For the purposes of this document, the following definitions and abbreviations apply.

2.1 Definitions

2.1.1 Allocated Function Data Memory

Function Data Memory (FDM) that is allocated for a particular instance of a function

Note 1 to entry:

See SNIA Computational Storage Architecture and Programing Model.

2.1.2 Computational Storage

architectures that provide Computational Storage Functions coupled to storage, offloading host processing or reducing data movement

Note 1 to entry:

These architectures enable improvements in application performance and/or infrastructure efficiency through the integration of compute resources (outside of the traditional compute & memory architecture), either directly with storage or between the host and the storage. The goal of these architectures is to enable parallel computation and/or to alleviate constraints on existing compute, memory, storage, and I/O.

Note 2 to entry:

See SNIA Computational Storage Architecture and Programing Model

2.1.3 Computational Storage Array

collection of Computational Storage Devices, control software, and optional storage devices

Note 1 to entry:

See SNIA Computational Storage Architecture and Programing Model.

2.1.4 Computational Storage Device

Computational Storage Drive, Computational Storage Processor, or Computational Storage Array

Note 1 to entry:

See SNIA Computational Storage Architecture and Programing Model.

2.1.5 Computational Storage Drive

storage element that provides Computational Storage Functions and persistent data storage

Note 1 to entry:

See SNIA Computational Storage Architecture and Programing Model.

2.1.6 Computational Storage Engine

component that is able to execute one or more CSFs

Note 1 to entry:

Examples are: CPU, FPGA.

Note 2 to entry:

See SNIA Computational Storage Architecture and Programing Model.

2.1.7 Computational Storage Engine Environment

operating environment for a CSE

Note 1 to entry:

Examples are: Operating System, Container Platform, eBPF, and FPGA Bitstream.

2.1.8 Computational Storage Function

specific operations that may be configured and executed by a CSE

Note 1 to entry:

Examples are: compression, RAID, erasure coding, regular expression, encryption.

Note 2 to entry:

See SNIA Computational Storage Architecture and Programing Model.

2.1.9 ***Computational Storage Processor***

device that provides Computational Storage Functions for an associated storage system without providing persistent data storage

Note 1 to entry:

See SNIA Computational Storage Architecture and Programing Model.

2.1.10 ***Computational Storage Resource***

resource available for a host to provision on a CSx that enables that CSx to be programmed to perform a CSF

Note 1 to entry:

A CSx contains one or more CSEs and each CSE executes one or more CSFs.

Note 2 to entry:

Examples are: CSE, CPU, memory, and FPGA resources.

Note 3 to entry:

See SNIA Computational Storage Architecture and Programing Model.

2.1.11 ***container***

software package that provides a secure environment to an application and host OSes

Note 1 to entry:

A container uses fewer resources and is lightweight compared to a conventional Hypervisor/VM configuration.

2.1.12 ***CSx name***

a string that identifies a CSx. This is returned in query requests (e.g., `csQueryCSxList`) and provided to the `csOpenCSx` function

2.1.13 ***filesystem***

software component that imposes structure on the address space of one or more physical or virtual disks so that applications may deal more conveniently with abstractly named data objects of variable size called files

2.1.14 Function Data Memory

device memory used for storing data that is used by the Computational Storage Functions (CSFs) and is composed of allocated and unallocated Function Data Memory

Note 1 to entry:

See SNIA Computational Storage Architecture and Programing Model.

2.1.15 host

computer system to which disks, disk subsystems, or file servers are attached and accessible for data storage and I/O

2.1.16 hypervisor

host OS with elevated privileges that works with hardware mechanisms such as Intel’s VT and VT-d technology and hosts VMs

2.1.17 Peer-to-Peer

data transfer directly between two devices that does not involve a host or host memory

Note 1 to entry:

A transfer directly between two CSDs. In PCIe devices, the transfer bypasses host memory.

2.1.18 P2P

Peer-to-Peer

2.1.19 PCIe®

Peripheral Component Interconnect Express is a high-speed serial computer expansion bus standard

2.1.20 string

a C language style string

Note 1 to entry:

A string is a sequence of characters that are treated as a single data item. A string is terminated by the null character '\0'.

2.2 Keywords

In the remainder of the standard, the following keywords are used to indicate text related to compliance:

2.2.1 ***mandatory***

a keyword indicating an item that is required to conform to the behavior defined in this standard

2.2.2 ***may***

a keyword that indicates flexibility of choice with no implied preference; “may” is equivalent to “may or may not”

2.2.3 ***may not***

keywords that indicate flexibility of choice with no implied preference; “may not” is equivalent to “may or may not”

2.2.4 ***optional***

a keyword that describes features that are not required to be implemented by this standard; however, if any optional feature defined in this standard is implemented, then that feature shall be implemented as defined in this standard

2.2.5 ***shall***

a keyword indicating a mandatory requirement; designers are required to implement all such mandatory requirements to ensure interoperability with other products that conform to this standard

2.2.6 ***should***

a keyword indicating flexibility of choice with a strongly preferred alternative

2.3 Abbreviations

AFDM	Allocated Function Data Memory
API	Application Programming Interface

CSA	Computational Storage Array
CSD	Computational Storage Drive
CSE	Computational Storage Engine
CSEE	Computational Storage Engine Environment
CSF	Computational Storage Function
CSP	Computational Storage Processor
CSR	Computational Storage Resource
CSx	Computational Storage Devices
DMA	Direct Memory Access
FDM	Function Data Memory
FPGA	Field-Programmable Gate Array
NVM	Non-Volatile Memory
P2P	Peer-to-Peer
SSD	Solid State Disk
VM	Virtual Machine

2.4 References

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

SNIA Computational Storage Artchitecture and Programming Model	available from https://www.snia.org/tech_activities/work

2.5 Conventions

Text in light blue indicates a common definition.

Text in pale blue indicates return values and parameters for functions.

Function names, data structure names, field names, and resource types, are denoted in `Courier New 12pt font`.

3 Computational Storage

These functionss provide definitions of functions to support the SNIA Computational Storage Architecture and Programming Model Specification.

As defined in the SNIA Computational Storage Architecture and Programming Model Specification, Computational Storage provides Computational Storage Functions coupled to storage, offloading host processing or reducing data movement.

Computational Storeage Devices (CSxes) as defined in the SNIA Computational Storage Architecture and Programming Model Specification include Computational Storage Processors (CSP), Computational Storage Drives (CSD), and Computational Storage Arrays (CSA) (see Figure 1)

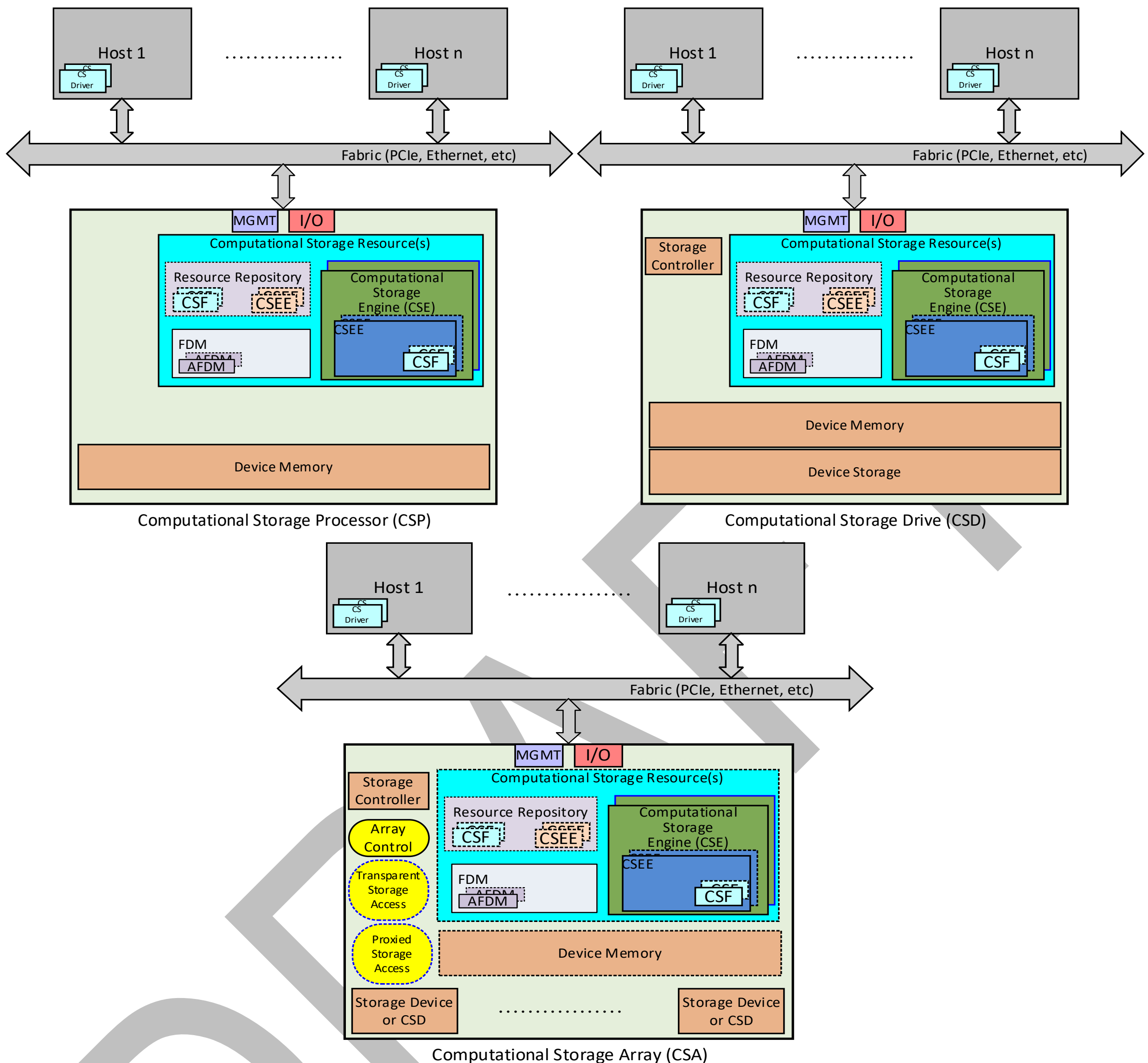

Figure 1: An Architectural View of Computational Storage

Additionally, a Computational Storage Function (CSF) is defined as a data function that performs computation on data as defined in the SNIA Computational Storage Architecture and Programming Model.

The following are examples of types of Computational Storage Functions:

a) Compression;
b) Encryption;
c) Database filter;
d) Erasure coding;
e) RAID;
f) Hash/CRC;
g) RegEx (pattern matching);
h) Scatter Gather;
i) Pipeline;
j) Video compression;
k) Data Deduplication; and
l) Large Data Set.

The following are examples of types of Computational Storage Engines:

a) Operating System Image;
b) Container Image;
c) Berkeley packet filter (BPF); and
d) FPGA Bitstream.

The SNIA Computational Storage Architecture and Programming Model describes Host Agents that are able to communicate with the device using a device driver and an interface (e.g., PCIe, Ethernet,). Host Agents are able to perform management, discovery, configuration, monitoring, operations, and security on the device. The fixed and programmable computational storage functions are programmable through a Host Agent using a well-defined interface.

4 API Overview

Computational storage uses Computational Storage Engines (CSEs) that are able to execute compute tasks that are typically run on a host CPU. These CSEs may use Function Data Memory (FDM) that is different from the host memory and from the memory for storing CSFs. A mechanism to transfer data to and from Allocated Function Data Memory (AFDM) is required. These data transfers are required for inputs and outputs to the CSE compute functions. Data transfers to AFDM may be from host memory and/or storage. There are specific functions that target these operations and interactions with the CSE. This section targets the usage of functions and how they are able to be used with CSEs for computational storage.

This standard defines a base set of functions that may be implemented in an API library as shown in Figure 2. Additional libraries are able to be built on this base set of functions. This version of the standard is tailored for a host orchestrated interface. Additional functions are required for a fully device managed interface.

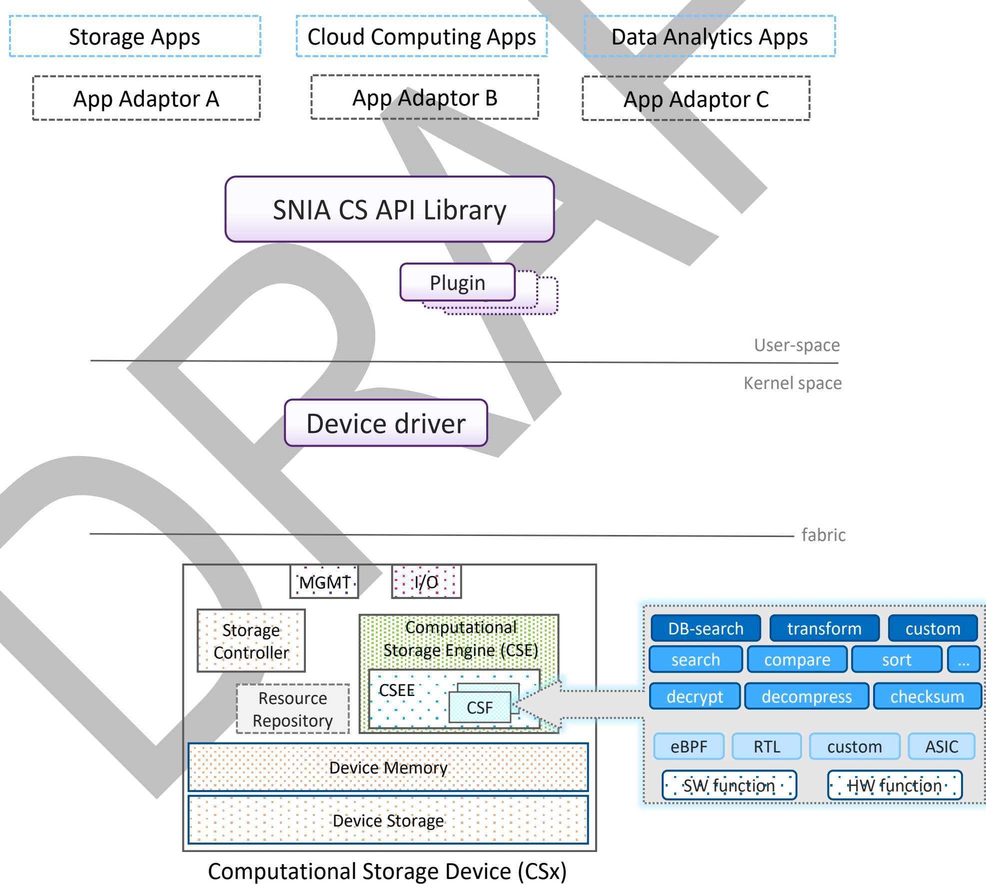

Figure 2: CS API Library

Although the functions have been tailored for a host-managed interface, they also apply to a device managed interface. In the device managed interface, the functions are implemented by the device. Discovery, access, allocation, and configuration of resources and all queued operations apply. Only the completion models may require host support to map them (e.g., callback vs. synchronous model).

If a CSF that is called by a Queue Batch Request or a Queue Compute Request attempts to access memory outside of AFDM then an Error In Executation status is returned.

The relationship between applications, the CS API, CSxes, and functions is shown in Figure 3.

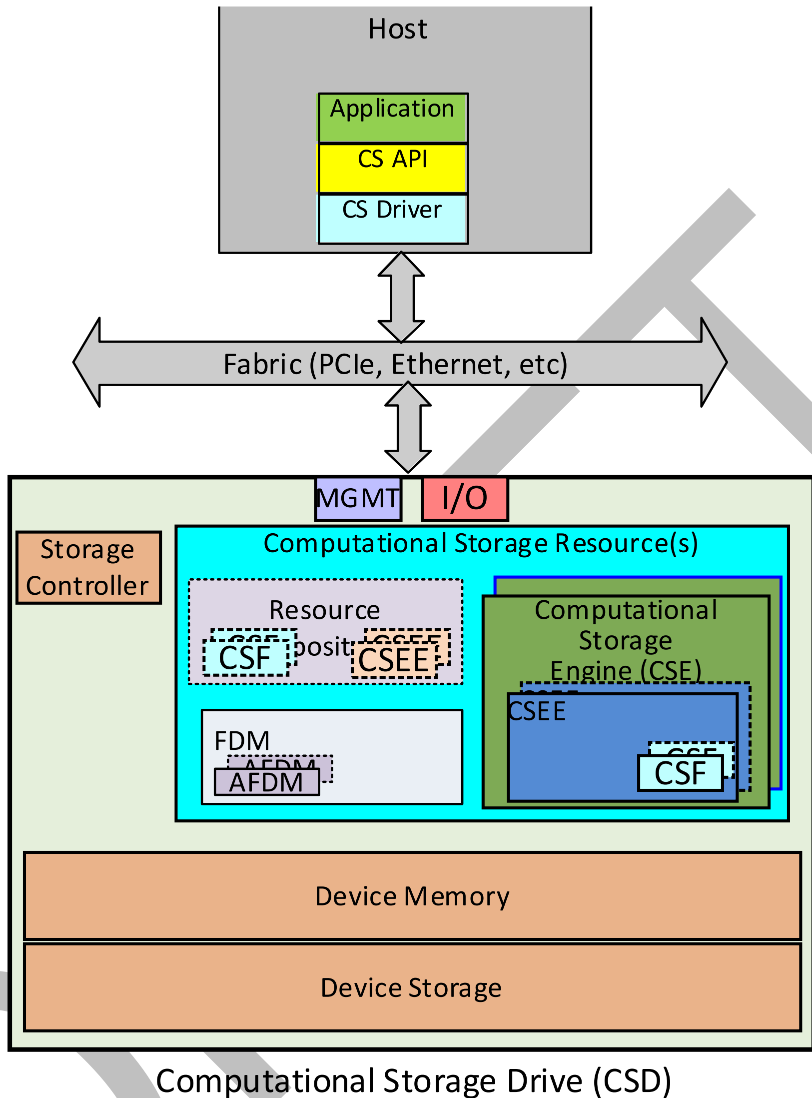

Figure 3: API Relationships

The discovery of resources and activation of resources is detailed in the SNIA Computational Storage Architecture and Programming Model. A brief overview is that once resources are discovered on a CSx, a Computational Storage Engine Environment (CSEE) is required to be activated on a CSE in order for that environment to be able to be used for CSFs. A CSF is required to be activated on a CSEE in order for that CSF to be able to be used for computation. In some implementations, activation of CSEEs and/or CSFs may be implicit. Additionally, the CSEE may be a logical construct that is associated with a CSE and is always activated by design.

4.1 Discovery and configuration

As shown in Figure 1, computational storage is provided by CSxes (i.e., CSDs, CSPs, and CSAs). Each of these may have their own configurations that may be specified prior to use (see 6.12.4). The CSx may be directly attached to the host or connected through a network or fabric. This specification is interface agnostic.

4.1.1 Discovery

4.1.1.1 CSx Discovery

CSxes may be discovered using the `csQueryCSxList()` function. The function returns a comma-separated list of CSxes. A CSx may also be discovered using the `csGetCSxFromPath()` function, where the path represents a device, directory or file.

Once a CSx is discovered, its resources may be queried with the `csQueryDeviceProperties()` function.

4.1.1.2 Discovery function

The `csQueryDeviceProperties()` function provides individual properties available at different resource levels (e.g., CSx, CSE, CSEE, FDM, and CSF). This function also provides details on the repository, activation states, and configuration. Figure 4 illustrates how this function may be applied at various resource levels by providing the resource identifier as the input. The engine type here is an abstract representation of the compute hardware resource and is specific to a device as provided by the vendor. engine type is uniquely identified by the field `CSETypeToken`. A vendor may choose not to expose engine differences.

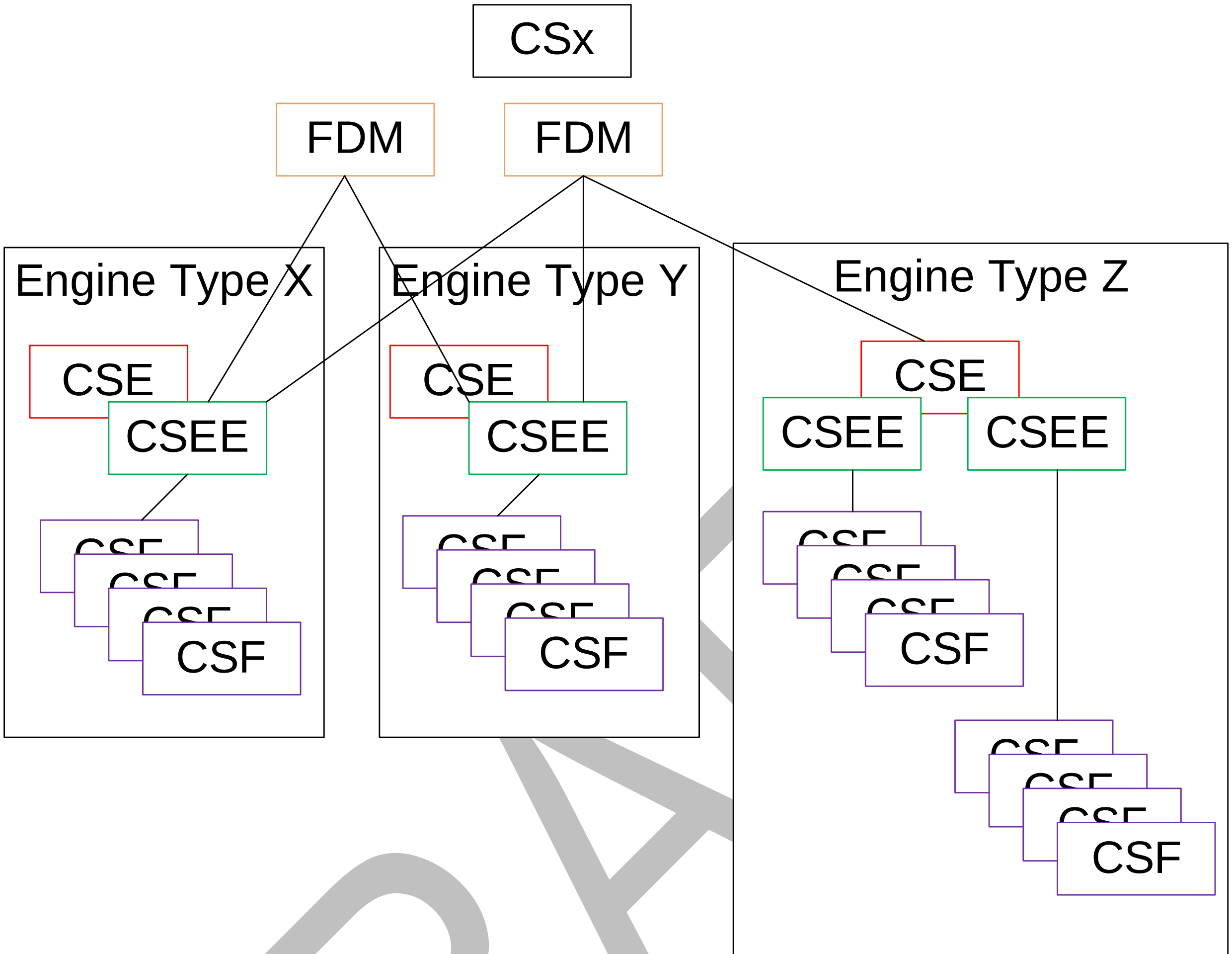

Figure 4: CSx Resource Overview

Figure 4 shows that a CSF is activated on a CSEE, a CSEE is activated on a CSE, and a specific FDM is associated with specific CSEs and therefore able to be associated with different CSEEs activated on different CSEs. AFDM for a specific CSEE is allocated out of FDM that is associated with that CSEE. The `csQueryDeviceProperties()` function takes the resource type as input and provides the properties of the resource as output. The `csQueryDeviceProperties()` function called with a resource type of `CS_CSx_TYPE` returns the `CSxProperties` data structure as output denoting hardware and software details of the CSx. Similarly, any of the other resource types may be provided as input to get the necessary outputs, as shown in Figure 4.

Table 2: Device properties by resource type

CS_RESOURCE_TYPE Input	Properties Output	Reference
CS_CSx_TYPE	CSxProperties	6.3.5.3.1.2
CS_CSE_TYPE	CSEProperties	6.3.5.3.1.5
CS_CSEE_TYPE	CSEEProperties	6.3.5.3.1.8
CS_FDM_TYPE	FDMProperties	6.3.5.3.1.12
CS_CSF_TYPE	CSFProperties	6.3.5.3.1.15
CS_VENDOR_SPECIFIC_TYPE	CSVendorSpecific	6.3.5.3.1.16

Additional details on the properties data structures and their sub-structures are provided in 6.3.5.3.1.

Figure 5 summarizes the functions required to discover and configure a CSx and its resources.

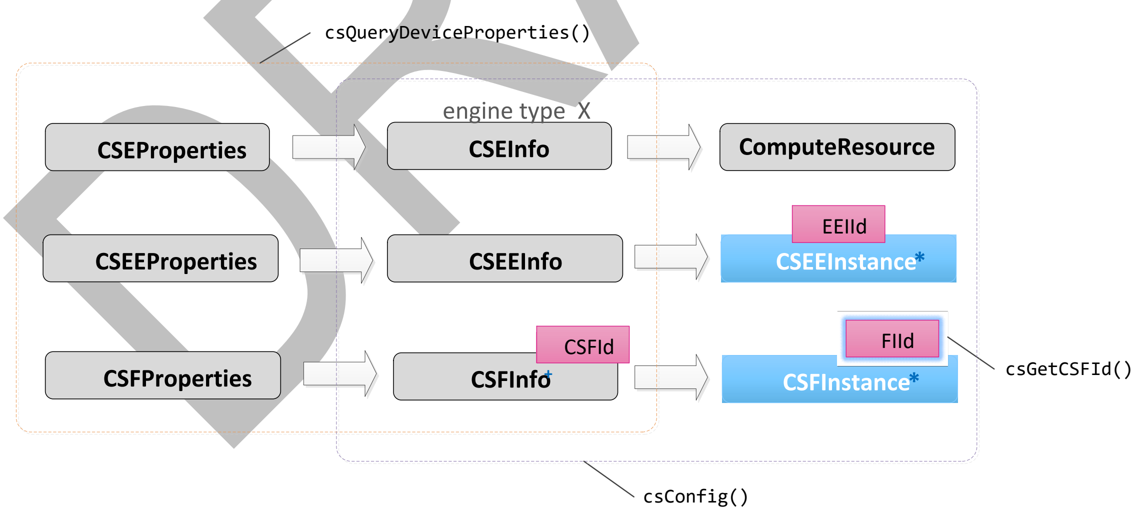

Figure 5: Function Mapping for Discovery and Configuration

4.1.1.3 CSF Discovery

A list of available CSFs is retrieved using the `csQueryDeviceProperties()` function using the `CS_RESOURCE_TYPE` enumerator `CS_CSF_TYPE`. Activated CSFs may be discovered using the `csGetCSFId()` function and the `csQueryCSFList()` function. Only activated CSF instances denoted by their FIId's are populated by the `csGetCSFId()` function and the `csQueryCSFList()` function.

4.1.1.4 Example discovery process

The following code example illustrates how the discovery functions may be applied. The CSx comma separated list is first parsed for individual CSx entries and then each entry is queried for each resource as shown below.

```
// query all available CSxes
status = csQueryCSxList(&len, listBuf);
token = strtok(listBuf, ",");
i = 0;
while (token != NULL) {
    status = csOpenCSx(token, NULL, &devArray[i]);
    // query for CSX properties
    status = csQueryDeviceProperties(devArray[i], CS_CSx_TYPE, &lenCSx, propCSx);
    if (status != CS_SUCCESS)
        ERROR_OUT("Query CSx properties error!\n");
    // query for CSE properties
    status = csQueryDeviceProperties(devArray[i], CS_CSE_TYPE, &lenCSE, propCSE);
    if (status != CS_SUCCESS)
        ERROR_OUT("Query CSE properties error!\n");
    // query for CSEE properties
    status = csQueryDeviceProperties(devArray[i], CS_CSEE_TYPE, &lenCSEE, propCSEE);
    if (status != CS_SUCCESS)
        ERROR_OUT("Query CSEE properties error!\n");
    // query for CSF properties
    status = csQueryDeviceProperties(devArray[i], CS_CSF_TYPE, &lenCSF, propCSF);
    if (status != CS_SUCCESS)
        ERROR_OUT("Query CSF properties error!\n");
    // loop through the whole list
    token = strtok(NULL, ",");
    i++;
}
```

4.1.2 Configuration

To be usable, a CSx is required to be configured. Once the CSx has been fully discovered, it may be configured using the `csConfig()` function. Configuration involves activation of the specific resource. This function takes the CsConfigInfo data structure as input to configure the specific resource.

Each resource is identified by an ID field in the associated data structure (e.g., the `CSEInfo` data structure for a CSE is identified by its unique CSEId).

4.1.2.1 Configuring a CSEE

A CSEE is required to be configured and activated before it may be used for CSF configuration and execution. A CSEE is activated by associating its CSEEId in CSEEInfo data structure with a CSEId in the `CSEInfo` data structures. More than one engine type may be paired with a CSEE by activating the ID pairs. Figure 6 depicts a CSEE being paired with a CSE whose Ids are provided as input, and on successful activation, return EEIId for the activated CSEE instance.

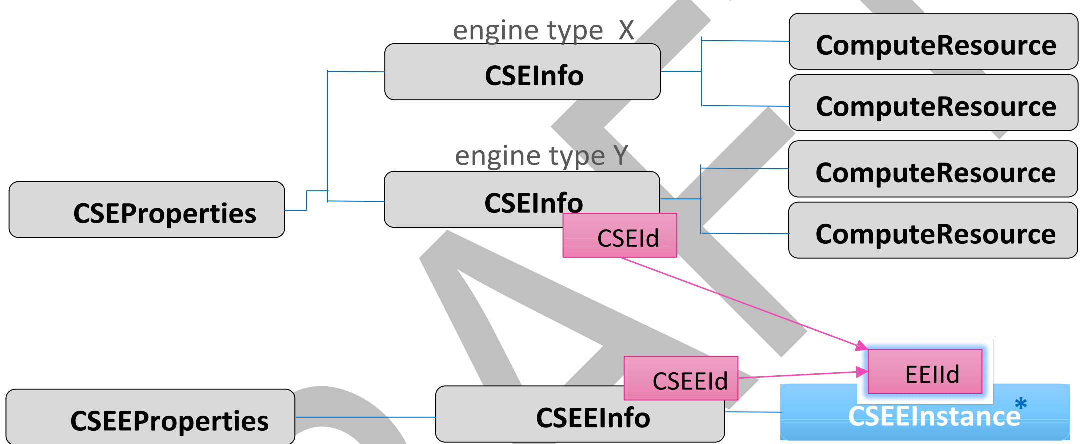

Figure 6: Activating a CSEE

The CSEE activated instance is available to be utilized for computational storage activities. The device in turn sets up internal configuration options and resources to bring the CSEE to the active state.

4.1.2.2 Configuring a CSF

Activation of a CSF involves associating the CSF with an active CSEE instance and one or more compute resources. As input, a previously activated CSEE instance EEIId is paired with a CSF using its CSFId and one or more compute resources represented by ERId. Figure 7 depicts the flow with the various inputs, and on successful activation, provides FIId for the activated CSF instance as output.

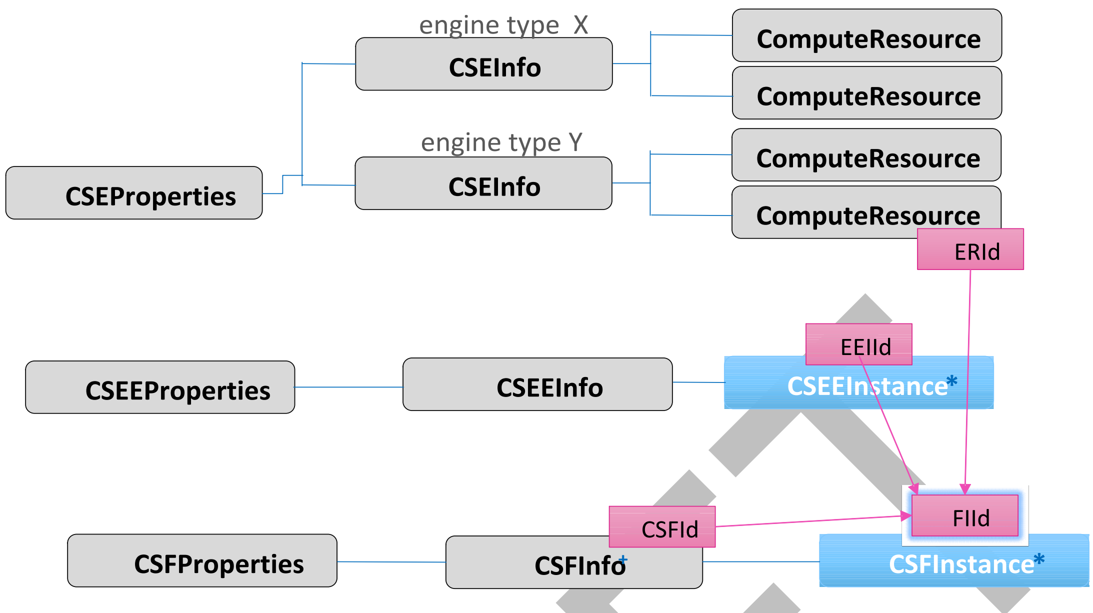

Figure 7: Activating a CSF

Activated resources consume device resources as part of their internal configuration. CSFs and CSEEs may also be deactivated using the `csConfig()` function. A resource is deactivated when it is no longer required or to release resources for other CSFs or CSEEs (e.g., a well-known algorithm-based CSF that is downloadable may be more performant than the vendor provided CSF that is built-in and therefore the built-in CSF may be deactivated to conserve device resources).

Additional details on configuration data structures are provided in 6.3.5.3.2.

4.2 FDM allocation

FDM is memory that is associated with the CSE and is separate from host memory but may be mapped to a host memory address. FDM is the memory that a computational storage function operates on. FDM may be exposed to the host (e.g., through a PCIe BAR) when direct attached and is not exposed at all for network attached usages.

There may be more than one FDM available to the device as shown in Figure 1. In a device that has more than one CSE, FDM may be configured for different accesses (e.g., one CSE may be configured to access all available FDMs while another CSE may be configured to access only one FDM). These details are discoverable through the device properties function. Since all FDM usage is based on the CSF and the CSE that can access that FDM, an application chooses the appropriate FDM while discovering CSFs through the `csGetCSFId()` function. This function provides a list of all FDMs that the CSF has access to, along with their access details, as specified in the `FDMAccess` data structure.

FDM is allocated and deallocated using the `csAllocMem()` function and the `csFreeMem()` function.

4.3 Compute types and execution

CSEs that are able to perform compute offload may be of various types (e.g., ASICs, FPGAs, and embedded CPUs). Execution of compute operations initiated by the `csQueueComputeRequest()` function or the `csQueueBatchRequest()` function are independent of the type of CSE. The type specific functionality of a CSE may be handled by a device driver whose implementation details may be abstracted at the function level.

For cases where a CSx does not exist and compute is conducted on the host CPU, the plugin framework may be utilized to provide similar functionality transparently so that the application does not have to change. Additional details on plugins are available in section 6.13.

4.4 Downloading functions

In certain CSEEs, CSFs are able to be downloaded. The `csCSFDownload()` function provides the mechanism for downloading CSFs to CSEEs with such capabilities. Following a download, the host may initiate a discovery to determine what CSFs are available.

4.5 Extending function support

A plugin provides the ability to extend the capabilities of the functions.

A plugin is a software entity that provides the data exchange between the abstracted CS functions and a device's specific interface. The data exchange is accomplished by having a mapping layer between these interfaces. A plugin may also abstract specific functionality for a device. Plugins also play a role in providing seamless access (e.g., local or remote connectivity using the same functions, supporting new features, or substituting/aiding in device feature support). Plugins may be applied at various places in a CS software stack implementation to provide features and to help support a common set of functions. Plugins are required to be registered first with the API library before they are able to be applied. The `csRegisterPlugin()` functionand the `csDeregisterPlugin()` functionare used to insert/remove plugin capability in the CS API stack.

4.6 Association of CSP and storage

Association between storage and a CSP is required for any device-to-device activity (e.g., peer-to-peer (P2P)) to function properly. With CSPs, the CSE is a free standing device where storage is separate. Without association, device-to-device operations have the possibility of failing since data may not be loaded or stored in the correct device. This problem becomes evident when more than one CSP is configured on the same system. The problem becomes severe when the host user application is not able to identify the association between these devices.

For PCIe implementations, issues that arise due to incorrect association result in data corruption, I/O failures in the case where the CPU prohibits access across root-complexes, and in virtualized environments where each device may get mapped in a way that has no co-relation at the PCIe bus level.

The mechanism to associate a CSP with one or more storage controllers is vendor specific and is out of scope for this document.

4.7 Function usage example

The following example (see Figure 8) illustrates the usage of CS functions for a typical flow for near data processing. In this example, the CSD provides decrypt function capability and does not expose FDM to the host. The steps below depict the individual items in Figure 8 for a CSD.

1) Host application allocates FDM input and output buffers for processing in CSx;
2) Data is next initiated to load from the storage device into input AFDM;
3) Data is loaded from the storage device into the AFDM by P2P transfer;
4) The decryption CSF is invoked to work on data in the AFDM;
5) The CSF posts the output data into the output AFDM buffer and notifies the application that the decryption is complete; and
6) The output results are copied from the output AFDM to host memory.

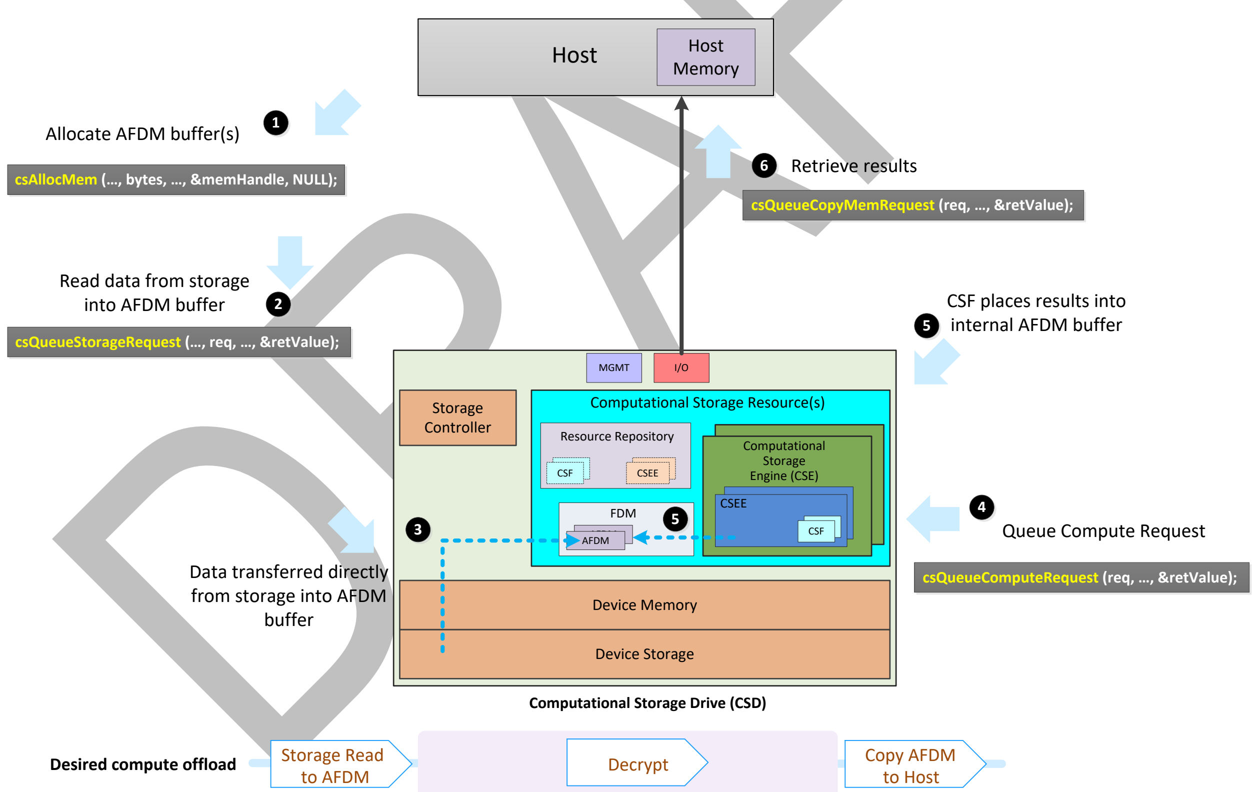

Figure 8: Example Function Flows

5 Details on common usages

5.1 FDM usage

A CSx has FDM that is allocated for a CSF to use for inputs and outputs. This memory is pre-allocated by the host application prior to its usage.

The `csAllocMem()` function and the `csFreeMem()` function are used to allocate and free FDM. This memory is allocated out of FDM and is referred to as AFDM.

CSxes may implement FDM in different ways. The function abstractions provide a transparent view of the FDM.

5.1.1 FDM usage example for CSD

This CSD example does not expose FDM to the host and hence all data transfers, while opaque, are described using the CS functions.

When the host allocates FDM buffers, they are referenced as AFDM. Once allocated, these AFDMs may be provided as input and output buffers for loading data from storage media, running compute functions with these data buffers, and copying data to and from host memory.

The host allocates the necessary amount of AFDM buffers with the `csAllocMem()` function.

The loading of storage media into the allocated AFDM is conducted by the `csQueueStorageRequest()` function.

Compute functions provided with these buffers are executed using the `csQueueComputeRequest()` function and the `csQueueBatchRequest()` function.

Data transfers between AFDM and host memory are conducted using the `csQueueCopyMemRequest()` function.

Key resources utilized by CSFs include compute and device memory. In this example, we use a generic CSF to describe compute and memory. For existing CSx architectures, memory usage is as follows:

a) Data transfer from host memory to FDM:
   A) Data for the CSF to work on; or
   B) Input parameters to the CSF;
b) Data transfer from FDM to host memory:
   A) Data that the CSF returns to host application; or
   B) Miscellaneous results (e.g., status and other variables);

and

c) Memory (that is outside of FDM) usage for CSFs:
    A) Internal device memory usage for CSFs during runtime not accessible by host (e.g., stack, scratchpad, operating system memory when the CSF is hosted by one device or local RAM for device-based functions).

In this architecture, the host pre-populates the data that the CSF has to work on (item a.A above) into the FDM. This is achieved by the device having the capability to transfer data directly between storage and FDM. For a CSx that does not contain storage such as a CSP, the host reads data into host memory from a storage device (e.g., SSD or CSD) and then copies that data to FDM on the CSP. These memory transactions involve DMA transfers through the fabric. This is because in this model, the CSFs have no direct DMA access to the host or peer device(s) and vice versa. Similarly, when the CSF has output data (item b.A above) stored in FDM that is required to be written to the media, the data is first DMAed to host memory and then written to the media. Each of these operations require 2 data transactions on the fabric, and in doing so, consume a part or all of the available bandwidth to the CPU. There is a high possibility of running into performance limitations when there are other similar devices populated and when network cards are also transferring data on the same fabric.

5.1.2 Allocating from FDM

FDM is allocated using the `csAllocMem()` function to provide memory for inputs and outputs of the CSF. FDM may or may not be visible in host address space depending on the CSx type. For example, Figure 8 depicts a CSD that does not expose FDM in the host's address space. The `csAllocMem()` function allocates FDM at a granularity as specified by the CSx. In addition to allocating FDM, this function also facilitates mapping that FDM into host's system address space, if the CSx supports this mapping.

5.1.2.1 When to map AFDM to a virtual address

Host address mapping should be requested as part of AFDM allocation, if AFDM is intended to be used for:

a) the OS filesystem/block subsystem to load data directly from the SSD utilizing the P2P protocol;
b) the OS filesystem/block subsystem to commit data directly from CSx to SSD using P2P; or
c) direct access from host application software.

The allocation request for mapping however depends on the ability of the CSx to have FDM exposed in host address space.

5.1.2.2 When not to map AFDM to a virtual address

AFDM should not request a virtual address pointer when allocated for the following usages:

a) AFDM is not exposed by the device to the host;
b) AFDM is used to transfer data from host memory as input to CSF for computation;
c) AFDM is used to collect results from CSF and subsequently copied back to host memory;
d) AFDM is used in batch requests;
e) When a CSx has large memory area to expose that may run into restrictions with the host systems BIOS;
f) When there are multiple CSxes and the additional exposed memory hits system BIOS limits; and
g) When the CSx is connected remotely.

For data transfers between host memory and device memory, the csQueueCopyMemRequest() function provides a mechanism for data transfer. In certain configurations (e.g., a virtualized configuration with a hypervisor), direct device memory access may provide unpredictable results and the DMA request may encounter errors (i.e., even though the memory is mapped with a virtual address, the memory access may still fail if accessed directly).

In these cases, device memory should be accessed through the device DMA engine using this function.

5.1.3 FDM to host memory mapping

FDM may be used as memory mapped to host address space or without a mapping. The device should be queried for its properties using the `csQueryDeviceProperties()` function to verify which modes memory access the device supports. The possible memory access types are:

a) memory exposed to host address space with mapping; or
b) memory not exposed to host address space.

5.1.3.1 FDM not exposed to host address space

In this example, FDM allocations with the `csAllocMem()` function does not request a virtual address pointer to be returned by setting the parameter `VaAddressPtr` to NULL. The device provides translations for such allocations internally for their memory locations. For this example, the function hides such details through the abstracted interface and provides the same definitions by skipping the mapping functionality. Remotely connected CSxes also adopt this usage model as they do not expose FDM as a virtual address to the local host.

Storage I/O to this type of FDM is achieved using the `csQueueStorageRequest()` function which facilitates the transfer of data from storage directly to FDM buffers where the transfers do not leave the device. Doing so may save host CPU usage, cache usage, memory usage, and fabric bandwidth. These savings translate into performance, latency, and power benefits.

5.1.3.2 FDM exposed to host address space

The function definitions support devices that also expose FDM to host address space. In this usage, a virtual address pointer is requested during allocation through the parameter `VaAddressPtr`. With CSxes that map FDM to host memory address space, it is possible to transfer directly between storage and the FDM using P2P. This saves on the additional hop to host memory, host CPU involvement, and external fabric transactions, in some cases.

The `csAllocMem()` function maps the AFDM to host's address space, if the device provides such an interface. With AFDM mapped to host address space, an application is able to perform P2P data transfers between SSD and AFDM using the filesystem.

5.1.3.2.1 *Using AFDM for P2P transfers*

As shown in Figure 9, devices operate with host CPU by exposing AFDM in the host's address space (e.g., the CSx makes its memory visible through a PCIe BAR). The CPU has full visibility of FDM in this system address space. Devices are able to transfer data to any physical address in host addressable memory.

AFDM is able to be used for P2P transfers as follows:

a) Host software allocates the required amount of FDM using the csAllocMem() function with the option of mapping to a virtual address. Memory should be allocated in a size that is aligned to the device and favorable of host software usage (e.g., in host OS page size increments which maps FDM to the host page boundary), where security protections are able to be enforced;
b) The mapped virtual address is able to be passed to a filesystem or block subsystem for read/write access. Before the AFDM buffer is provided as input to the filesystem, the application is required to ensure that no buffering occurs in the I/O request. This may be achieved by disabling I/Os from being cached by the OS. For filesystems, the file should be opened with the O_DIRECT flag so no buffering occurs and the I/O is directly submitted to the OS block layer. If not, the

results are indeterminate since data may be directly passed to the CSx and any caching layers in between may prevent this;

c) Memory passed to the SSD is required to start at the minimum offset supported by the block device. This is 4KB for all modern SSDs; and

d) The SSD DMAs data to an address that resides on the CSx. P2P is complete when the I/O request is complete and signaled back to the host as part of the normal I/O operations. The DMA transfer that occurred between the SSD and the AFDM does not involve the external fabric if both devices are within the same device enclosure. This action saves fabric bandwidth and associated latencies with the I/O. For user space filesystems and block level accesses, the virtual address returned in step b) is passed directly through an ioctl call to the device driver. Translations may be required from the appropriate filesystem to describe the I/O request at the block level;

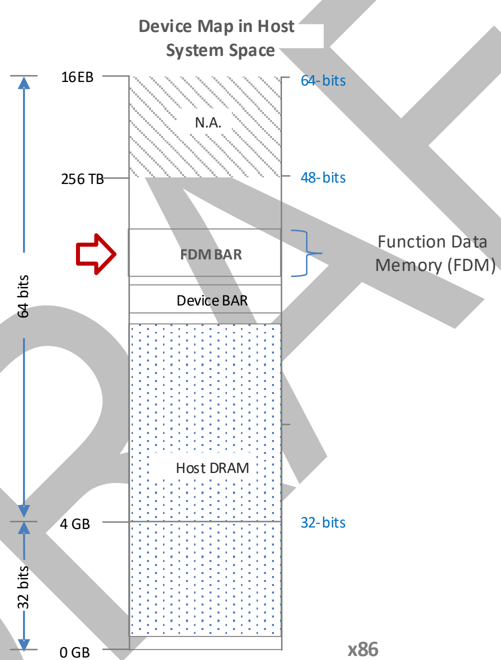

Figure 9: System Memory Map

e) The application then invokes the CSF to act on the data transferred. The CSF has local access to the data transferred since that data is in AFDM; and

f) When compute is complete, the CSF passes the data back to the application memory either through the csQueueCopyMemRequest() function or committing the data directly to SSD as in step 4.

Even though data movement is offloaded from host memory, the host CPU is still involved in the orchestration of data, as this is where the application resides.

There are three key advantages with the peering approach:

a) Reduction of PCIe bus bandwidth utilization;
b) Reduction in CPU utilization due to reduced memory copies; and
c) Reduction in host memory utilization.

5.1.4 Copy data between host memory and AFDM

Data transfers between host memory and AFDM requires only the `csQueueCopyMemRequest()` function.

This function takes data transfer direction as part of the request, as shown in Figure 10.

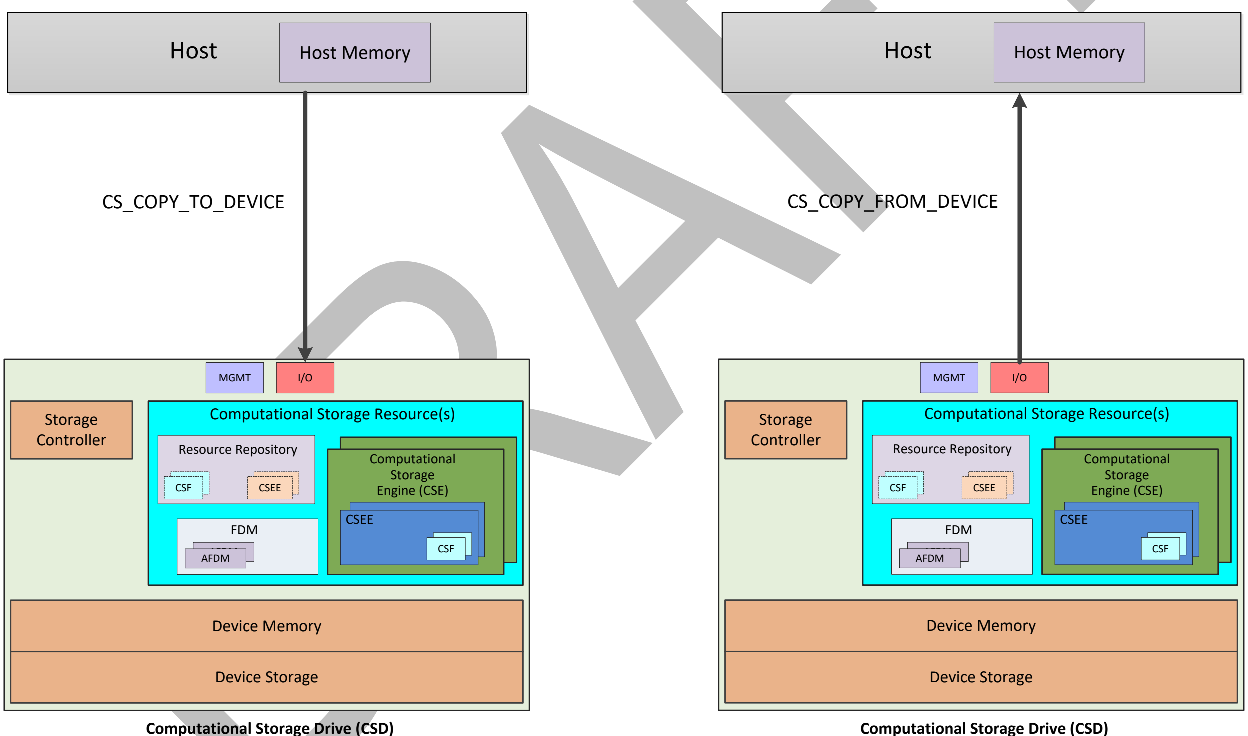

Figure 10: Example Data Transfers Between AFDM in a CSx and Host Memory

5.2 Scheduling compute offload jobs

Scheduling compute offload is done using the `csQueueComputeRequest()` function. This functiontakes as input the CSF to which a job should be queued along with its arguments. The number of arguments and their values should match the definition of the CSF as the CS API library will not enforce these and the behavior may be undefined.

An advanced method of queuing jobs is batching multiple requests together using the `csQueueBatchRequest()` function. This function allows multiple jobs to be batched together as one request.

Compute offload jobs require input and produce output (see Figure 11). Each of these jobs require a job request.

Figure 11: Job Processing

Figure 11 summarizes job processing for input, compute and output.

Table 3: Job request processing

<table>
<tr><th>Job</th><th>Details</th></tr>
<tr><td>Input</td><td>Provides input to a compute job. Input to a compute job may be provided in two ways:
<table>
<tr><th>Input method</th><th>Related function</th></tr>
<tr><td>Storage</td><td>a) Use file system calls with device memory mapped to host; and
b) Use the csQueueStorageRequest() function with type option CS_STORAGE_LOAD_TYPE</td></tr>
<tr><td>Host memory</td><td>Use the csQueueCopyMemRequest() function or the csQueueBatchRequest() function with option CS_COPY_TO_DEVICE</td></tr>
</table>
</td></tr>
<tr><td>Compute</td><td>The actual compute job may be scheduled to run in the following ways:
<table>
<tr><th>Method</th><th>Related function</th></tr>
<tr><td>Single or batch request</td><td>Use the <code>csQueueComputeRequest()</code> function for a single request or the <code>csQueueBatchRequest()</code> function for batch request.</td></tr>
</table>
</td></tr>
</table>

Table 3: Job request processing

<table>
<tr><td>Output</td><td>Provides output from a compute job. Output from a compute job may be received in two ways:
<table>
<tr><th>Input method</th><th>Related function</th></tr>
<tr><td>Storage</td><td>a) Use file system calls with device memory mapped to host; and
b) Use the csQueueStorageRequest() function with type option CS_STORAGE_STORE_TYPE</td></tr>
<tr><td>Host memory</td><td>Use the csQueueCopyMemRequest() function or the csQueueBatchRequest() function with option CS_COPY_FROM_DEVICE</td></tr>
</table>
</td></tr>
</table>

5.2.1 Batching requests

The `csQueueBatchRequest()` function is an advanced queuing mechanism that minimizes the interactions between host software and the device by optimizing the input(s) and output(s). A batch request is a sequence of requests that are executed without application interaction. In the sequence of requests, requests are linked such that one request is required to complete before the request that follows it starts. It is useful in cases where the work required to be performed by the CSx is required to be done in a particular order with a set of jobs. These could be serialized jobs, parallelized jobs, or hybrid ordering of jobs (see Table 4) that may be queued to a CSx. Jobs may be combined into a single batch request and submitted by the application at one time and get notified of a completion response only after all of the jobs are done.

Table 4: Batch Mode

Batch mode	Details
Serial	A batch request that has more than one request that is executed in pipeline mode, where, the next job will not start until the current job is complete. Since dependency is explicit, only the request details are necessary to execute the batch request. Batch requests are listed serially using the helper functions. Individual functions that are able to be batched serially are the `csQueueStorageRequest()` function, the `csQueueComputeRequest()` function, and the `csQueueCopyMemRequest()` function.
parallel	In this mode of execution, the intended purpose is to break down a larger request into smaller jobs and execute them independently. There is no dependency on any of these parallel jobs within the request and they may all start together at the same time. The `csQueueStorageRequest()` function, the `csQueueComputeRequest()` function, or the `csQueueCopyMemRequest()` function are able to be batched in a single request to execute in parallel. These functions may also be mixed together and also run in parallel. The supporting hardware is required to support the required parallelism for this batch operation to execute as intended.
hybrid	In this mode, complex and nested operations are able to be performed with the batch request. The `csQueueStorageRequest()` function, the `csQueueComputeRequest()` function, and the `csQueueCopyMemRequest()` function are able to be batched in a single request to execute in batch mode. The sequence of

Table 4: Batch Mode

	requests may be included as single requests or as a series of nested graph operations.

Batch requests are built by first allocating a batch request using the csAllocateBatchRequest() function. This function returns a handle for the batch request. Job entries may then be added to this batch request using the csConfigureBatchEntry() function. Batch requests are then queued to the device using the csQueueBatchRequest() function.

Table 3 describes how jobs are associated within a batch request using the `Before` and `After` parameters in the csConfigureBatchEntry() function (see **Error! Reference source not found.**).

Table 5: Batch Entry Association

Before	After	Details
0	0	This entry is not associated with any existing entries in the batch request
0	Non-zero	Places the entry immediately following the entry indexed by After
Non-zero	0	Places the entry immediately preceeding the entry indexed by Before
Non-zero	non-zero	Places the entry immediately preceeding the entry indexed by Before and immediately following the entry indexed by After.

If a batch entry association creates a loop or any association that the device is not capable of, then the request to add the batch entry to the batch handle generates an error. Batching requests by using this function helps an application to pipeline multiple jobs by their dependencies, reduce host CPU usage, reduce latencies by having less host context switches, and providing a more optimized execution path. Most computation jobs tend to have a combination of more than one queued job to complete the required task in a combination of input, compute, and output jobs. Batching requests may or may not be supported in hardware. For cases where batching is not supported in hardware, an underlying software implementation of the function may provide similar functionality. Batch request functionality is able to be discovered using the `csQueryDeviceProperties()` function.

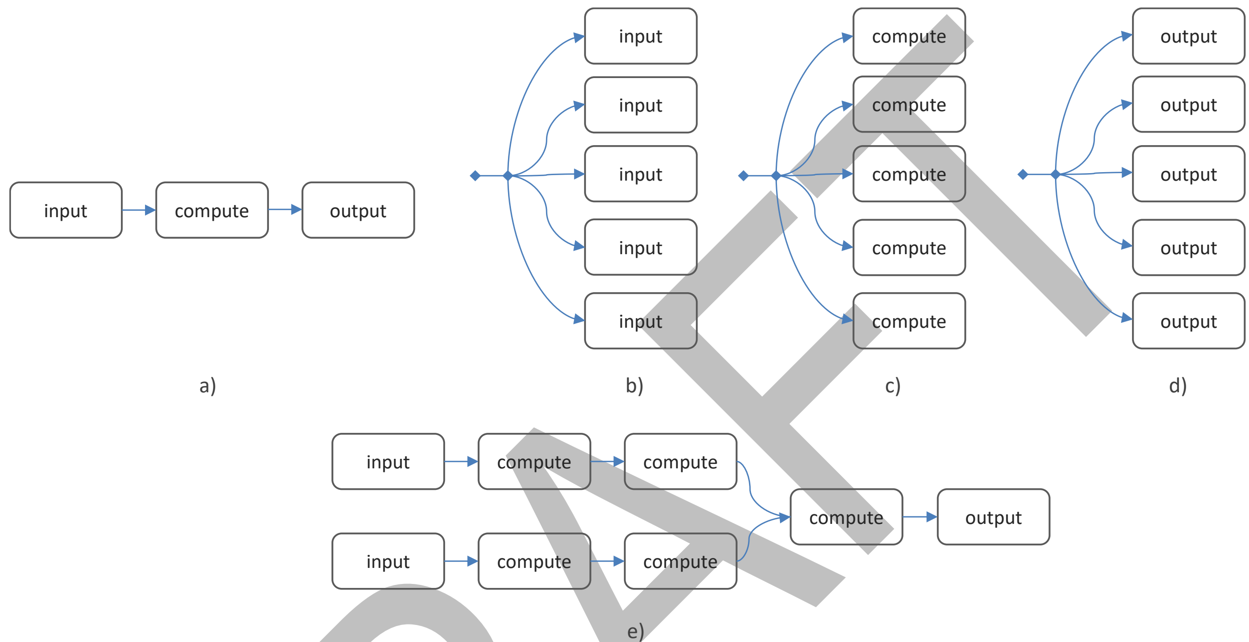

Figure 12: Batch Requests

Figure 12 illustrates different types of batch requests. In option a, a serialized notation of job requests using the batching option is shown. In this option, input is the first job and on completion, provides data to the compute job. On completion of the compute job, the results are provided to the output job, which satisifies the serialization and dependency requirements. Options b, c, and d illustrate parallel operations of job processing for input, compute, and output respectively. Option e represents a more complex batch request where there are more inputs and more compute requests in one batch request. This option also exhibits parallelism and dependencies from the previous job, as applicable. The usage of each job type is defined in Figure 12.

Here are a few illustrative examples on how multiple job requests may be scheduled with one request.

5.2.1.1 Serialized operations example

Serialized operations, as shown in Figure 13, involve dependencies, where the output of the previous job is the input to the next job. Instead of submitting each of these jobs individually, the user is able to create a batch request and post them at one time and get the results after the last job has completed. On the CSx, the requests will be processed serially and will not interrupt the user on completion of each job in the batch.

Figure 13: Serialized Operation

A serial batch request presents jobs as an array that specifies the required order. Serial batch request implies a dependency between the previous job and the next job and does not require additional dependency details as a hybrid operation does (see 5.2.1.3).

In Batch Example 1 (see Figure 14), data is first copied from host memory to device and compute offload work is scheduled after the copy is done. The next operation does not start before the previous operation is completed.

Figure 14: Batch Example 1

Batch Example 2 (see Figure 15) is the same as the Batch Example 1, with the addition of copying the results back to host memory. This example demonstrates an input job, a compute job and an output job.

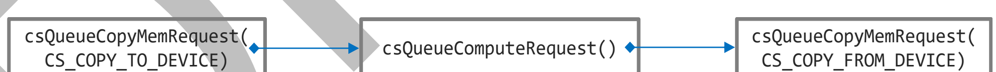

Figure 15: Batch Example 2

Batch Example 3 (see Figure 16) is a typical flow that manipulates stored data and provides the output back to host.

Figure 16: Batch Example 3

In Batch Example 4 (see Figure 17), the output of a compute request becomes the input to the next compute request.

Figure 17: Batch Example 4

For additional details, see sample code in section A.3.

5.2.1.2 Parallelized operations examples

Parallelized operations apply to jobs that are required to be done by multiple CSEs at the same time in a distributed manner. The ability to do so is required to be supported by the CSE.

Parallelized Operation Example 1 (see Figure 18) shows six compute jobs that are initiated at the same time and their completion results are conveyed back after all of them are completed. This type of scheduling and completion greatly simplifies the application orchestration tasks on the host side.

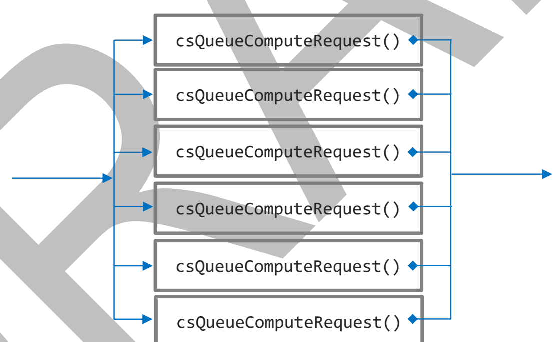

Figure 18: Parallelized Operation Example 1

In Parallelized Operation Example 2 (see Figure 19), data results may have been completed in AFDM by many CSFs or the results may be fragmented and ready for the host. The batch request helps in collating the results back to the host in a manner similar to scatter gather lists.

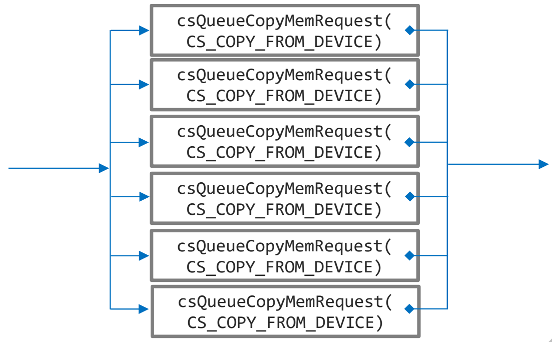

Figure 19: Parallelized Operation Example 2

With some CSx implementations, DMA copy operations may be more efficient if multiple requests are collapsed together with a single request for best performance.

The parallelized operations apply very well with distributed compute usages not only for single CSEs but also for multiple CSEs and may be more optimal from the execution point of view. As shown in the above two examples, the same operations may be queued to two different CSEs with a single function request. This may provide interesting and powerful application outcomes.

For additional details, see sample code in section A.3.

5.2.1.3 Hybrid operations examples

Hybrid scheduling operations are able to be employed when the current job's input depends on the previous job's output to complete. These may be in any order and nested too. Here are some examples of the combinations.

a) A previous serial/parallel job's output is the input to the next serial/parallel job;
b) A previous storage job's output is the input to the next serial/parallel job; and
c) A previous data copy job's output is the input to the next serial/parallel job.

Each of these use cases has a serialization step between the completion of one operation and execution of the next operation. A dependency exists that one operation has to complete to provide the data required by the subsequent operation. The use case where a serial job depends on a previous serial job is not covered above since that case may be handled by serialized operations as listed in section 5.2.1.1. There may also be paths where data dependency does not exist. This may be the case which has multiple inputs at the start of the batch request and where each request may take a different path. The Hybrid Batch Example (see Figure 20) shows such a case. This is also depicted in scenario e) of Figure 12.

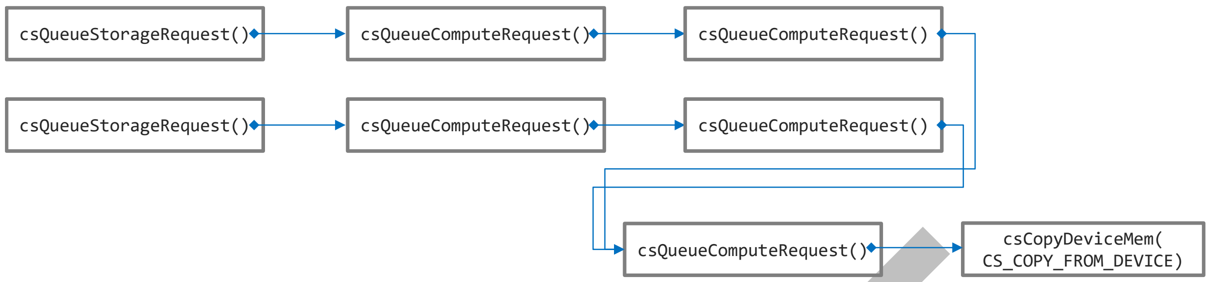

Figure 20: Hybrid Batch Example

Since a data dependency exists and the data resides in device space, the data is able to be provided as an input to enable hybrid mode using the `csQueueBatchRequest()` function. Batch requests in hybrid mode may take dependencies into account as part of execution. Serial and parallel requests by design are assumed to follow a specific flow and no additional information on dependency may be followed in the execution path.

Scheduling hybrid batch operations is possible using the `csQueueBatchRequest()` function with additional parameters. The additional batch functions define the dependencies by resource type and provides details on what the current request depends on to complete before the function is able to start. Using these dependencies, complex operations as listed in the combinations above are able to be performed by queuing them in advance and allowing the subsystem to take care of the executions and order. This may also be handled directly in the device or by the software framework without application intervention.

5.2.1.4 Building a batch request

In the example in Figure 20, each request represents a node in the batch of requests with `ReqType` specifying the operation for that node. Each request is added to the batch request with the `csConfigureBatchEntry()` function. The function takes the previously created `BatchHandle` and `Req` (i.e., the batch entry to be added) as inputs. Additional inputs `Before` and `After` specify the position of the request to be placed in the batch of requests. The function on successful addition of the batch entry returns `Curr` (i.e., the index of the request within the batch request). The value of `Curr` is unqiue only within that batch request. The value of `Curr` is a positive integer.

Using the example in 5.2.1.3, Figure 21 represents each batch entry by n1 to n8. Each of these entries will have its own `Req` that specifies the requests parameters.

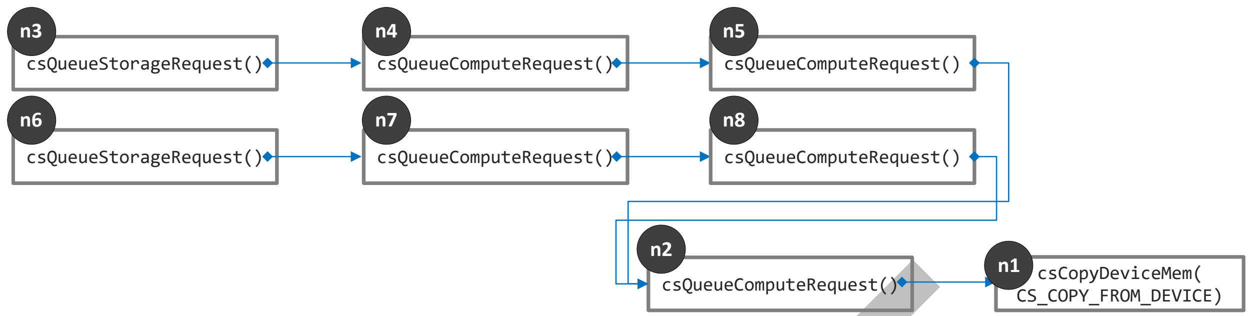

Figure 21: Building a Batch Request

The example in Figure 21 is able to be created using the parameters in Table 4 and returning the current entry index in Table 4:

Before	After	Current	Comments
0	0	n1	New entry not associated with any others.
n1	0	n2	New entry before n1.
0	0	n3	New entry not associated with any others.
0	n3	n4	New entry after n3.
n2	n4	n5	New entry after n4 and before n2. This entry links the two sequences.
0	0	n6	New entry not associated with any others.
0	n6	n7	New entry after n6.
n2	n7	n8	New entry after n7 and before n2. This entry links the two sequences and creates the requirement that n5 and n8 complete prior to execution of n2.

5.2.2 Optimal scheduling

Batch based scheduling requests provide optimal I/O flows to and from CSFs. The scheduling of compute and data movement internally utilize the most efficient path available through the compute offload device. No separate calls are necessary to prepare for the compute.

Some attention has to be placed on the CSE if more than one CSF is queued for execution at the same time. If multiple CSFs are queued on a CSE, then function grouping is required to be used to provide hints during scheduling.

For a batch request where compute output from multiple batch requests is required to be aligned, the scheduler on the CSX should manage the transitions between different batch requests for efficient execution.

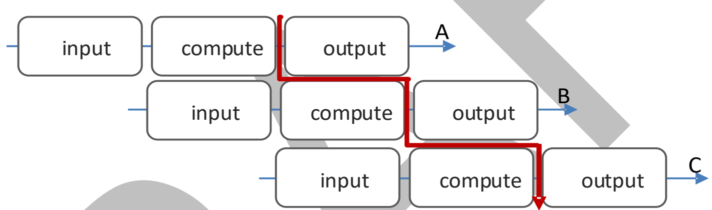

Figure 22: Optimal CSF Scheduling

Figure 22 depicts a shared compute resource, indicated by the bold line, being utilized efficiently with minimal idle time. A, B, and C are separate batch request executions that use a single CSF. For this example, start of execution also depends on when the previous input request completes.

5.3 Working with CSFs

CSF functionality depends on its CSE implementation. Since CSEs' types may be built differently from one another, a CSF for one type of CSE may not have similar characteristics to a CSF for another type of CSE. A CSF for one CSE type may not work with another CSE type. Some CSFs may be able to execute multiple instances from one image while others may require their own image instance to run.

A CSx may be preloaded with zero or more CSFs by the manufacturer. CSFs may also be downloaded. These two sources for CSFs and the available count are dependent on the implementation of the CSx resources, CSEE resources, and CSE resources. CSFs may be downloaded using the `csCSFDownload()` function.

A CSF may be represented by a name such as compress, checksum etc. Since a name string may not be able to uniquely represent a CSFs implementation, a global unique identifier is also supported. This identifier may provide a standardized representation of the CSF's algorithmic implementation.

A CSF by default resides in the resource repository. CSFs are required to be configured and activated before they can be executed. This involves associating the CSF with the correct CSE and CSEE environment. Configuration and activation is achieved using the `csConfig()` function. They are discovered for this step using the `csQueryDeviceProperties()` function. Both of these functions are privileged operations and are used to setup a CSx.

Activated CSFs are discoverable by non-privileged users using the `csGetCSFId()` function which returns details on one or more CSFs in the `CSFIdInfo` data structure. These details include a `CSFId` that may be used to execute the CSF, relative performance and power details, which may be used to choose a CSF from the list and a count of the instances available for execution.

Executing a CSF is done using the `csQueueComputeRequest()` function. An advanced version of execution may also be achieved using the `csQueueBatchRequest()` function, which facilitates batching a sequence of operations.

To save resources, a CSF instance, if not utilized, may be deactivated using the `csConfig()` function. A CSF that was previously downloaded may be unloaded using the `csCSFDownload()` function.

5.4 Completion models

Storage, Memory Copy, and Compute requests use a queued I/O model, where the request may be queued. These requests have three different options to complete the request as shown in Table 6. The requests may be queued for synchronous or asynchronous completions.

With the synchronous completion model, the request does not return until it is completed by the API library.

With an Asynchronous completion model, the request may be queued with a callback function or with an event. The callback function will be notified asynchronously in an arbitrary thread context when the request completes. With events, the user may poll using the `csPollEvent()` function in the caller's context and when ready to process the completion. The I/O for both asynchronous completion types gets a completion back only when the request at the device is complete.

Table 6: Completion Models

Completion Model	Inputs	Description
Synchronous	Context = NULL CallbackFn = NULL EventHandle = NULL	This is a blocking model, where the submitted request will not return to caller until complete.
Asynchronous Callback	Context = <User Context> CallbackFn = <User Callback function> EventHandle = NULL	This is a non-blocking model, where the user callback function is invoked when the requested I/O is complete.
Asynchronous Event	Context = <User Context> CallbackFn = NULL EventHandle = <User Event handle>	This is a non-blocking model, where the user event is signaled when the requested I/O is complete. The user is able to poll the event handle for completion status to change from CS_QUEUED.

6 CS function Interface definitions

CS functions enable interfacing with one or more CSEs and provide near storage processing access methods. Definitions will be provided in the following file

```
#include "cs.h"
```

This C programming language header file contains structures, data types and interface definitions. The associated interface definitions for the functions will be provided as a user space library. The details of the library are out of scope for this document.

6.1 Function access and flow conventions

The function definitions listed in this section use the following convention for handles. Handles have very specific usage. Only one handle is accepted per task as the main input and additional handles may be referenced as arguments.

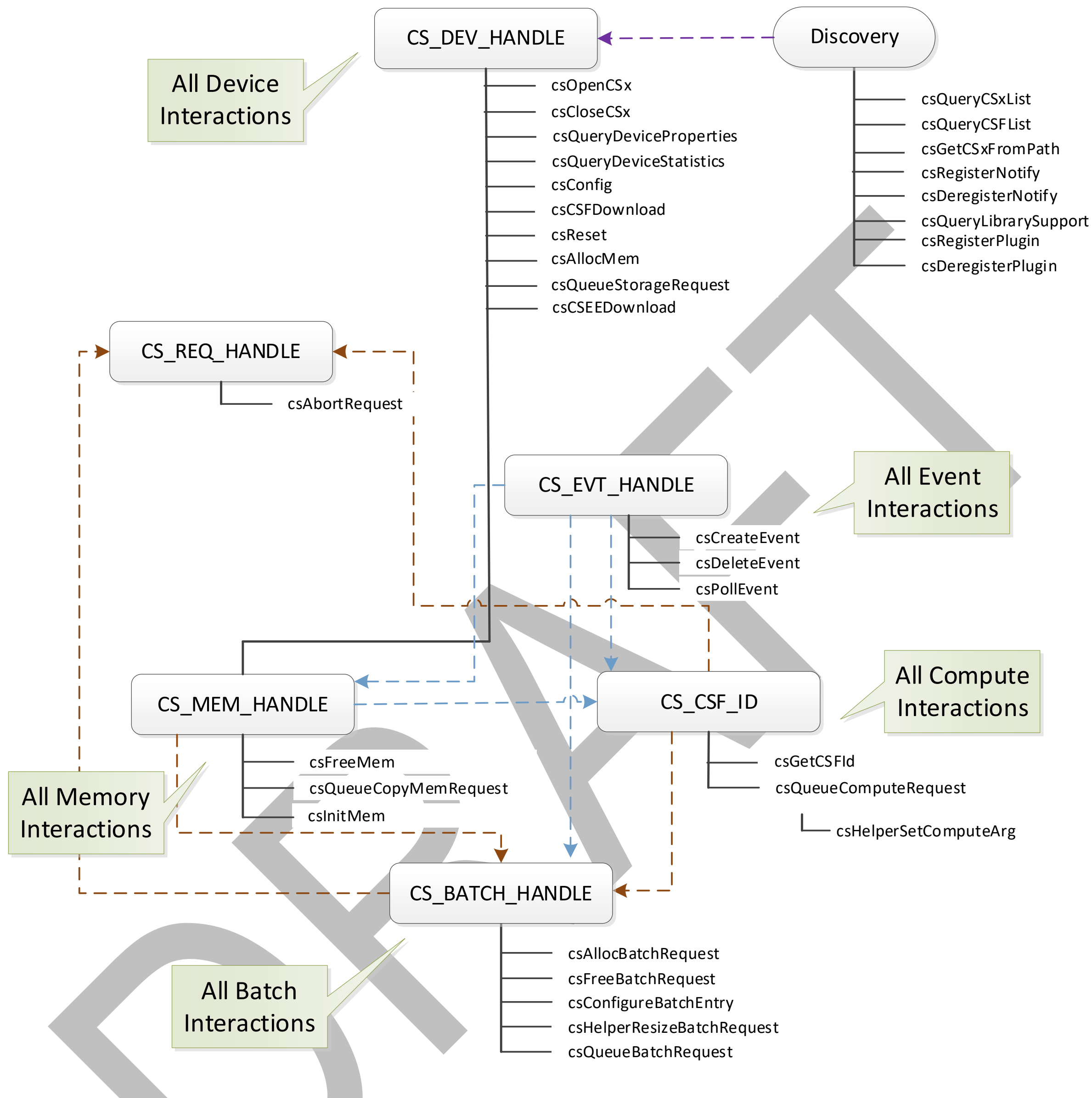

Figure 23: Function Access Flows

6.2 Usage overview

The CS function interface to applications is able to be broken down by functionality into the sections as defined in Table 7.

Table 7: CS Function Matrix

Functionality	Functions
Device Discovery - Identify CSxes - Identify CSFs - Identify CSx associated with Storage device	csQueryCSxList() csQueryCSFList() csGetCSxFromPath()
Device Access - Open/Close CSx device for access	csOpenCSx() csCloseCSx()
FDM management - Allocate/Deallocate FDM	csAllocMem() csFreeMem() csInitMem()
Storage IOs - Issue read/write IOs from/to Storage	Use filesystem with FDM and initiate P2P csQueueStorageRequest()
CSx Data movement - Transfer data between device memory and host memory	csQueueCopyMemRequest()
CSF access and scheduling - Schedule CSF on device	csGetCSFId() csAbortRequest() csQueueComputeRequest() csHelperSetComputeArg() csQueueBatchRequest()

Table 7: CS Function Matrix

	csAllocBatchRequest() csFreeBatchRequest() csConfigureBatchEntry() csHelperResizeBatchRequest()
Device Management - Query device properties and capabilities - Manage device functionality	csQueryDeviceProperties() csQueryDeviceStatistics() csCSFDownload() csCSEEDownload() csConfig() csReset() csRegisterNotify() csDeregisterNotify()
Event Management - Create/delete events for completion processing	csCreateEvent() csDeleteEvent() csPollEvent()
Library Management - Query library support - Manage library interfaces to support functions	csQueryLibrarySupport() csRegisterPlugin() csDeregisterPlugin()

6.3 Common definitions

6.3.1 Character arrays

All strings are null terminated. Since a string is null terminated, the maximum number of non-null characters in the character array is one less than the size of the character array.

The null termination character is not included in the string length.

6.3.2 Data types

Name	Description
s8	Signed 8-bit data; used as input to functions and arguments
u8	Unsigned 8-bit data; used in arguments scheduling a CSF
f8	Float 8-bit data; used in arguments scheduling a CSF
s16	Signed 16-bit data; used as input to functions and arguments
u16	Unsigned 16-bit data; used in arguments scheduling a CSF
f16	Float 16-bit data; used in arguments scheduling a CSF
s32	Signed 32-bit data; used as input to functions and arguments
u32	Unsigned 32-bit data; used in arguments scheduling a CSF
f32	Float 32-bit data; used in arguments scheduling a CSF
s64	Signed 64-bit data; used as input to functions and arguments

u64	Unsigned 64-bit data; used in arguments scheduling a CSF
f64	Float 64-bit data; used in arguments scheduling a CSF
u128	Unsigned 128-bit data; used in arguments scheduling a CSF

6.3.3 Status values

One of the values in Table 8 is returned by the interface functions and is classified under `CS_STATUS`. See section 6.3.5.2.18 for details on values.

Table 8: Status Value Definitions

Status Value Definition	Description
CS_COULD_NOT_MAP_MEMORY	The requested memory allocated was not able to be mapped
CS_COULD_NOT_UNMAP_MEMORY	Memory previously mapped was not able to be unmapped for AFDM
CS_DEVICE_ERROR	The device encountered an error and was not able to make progress
CS_DEVICE_NOT_AVAILABLE	The CSx was unavailable
CS_DEVICE_NOT_READY	The device was not ready for any transactions
CS_DEVICE_NOT_PRESENT	The specified device was not present
CS_ENTITY_NOT_ON_DEVICE	The entity does not exist on requested device

Table 8: Status Value Definitions

CS_ERROR_IN_EXECUTION	There was an error that occurred in the execution path
CS_FATAL_ERROR	There was a fatal error that occurred
CS_HANDLE_IN_USE	The requested handle was already in use
CS_INVALID_HANDLE	An invalid handle was passed
CS_INVALID_ARG	One or more invalid arguments were provided
CS_INVALID_CSF_ID	The CSF identifier specified was invalid
CS_INVALID_CSF_NAME	The CSF name specified does not exist or was invalid
CS_INVALID_FDM_ID	The FDM identifier specified was invalid
CS_INVALID_GLOBAL_ID	The Global Identified specified was not valid
CS_INVALID_ID	The specified input ID was invalid and does not exist
CS_INVALID_LENGTH	The specified buffer was not of sufficient length
CS_INVALID_OPTION	An invalid option was specified
CS_INVALID_PATH	No such device, file, or directory exists
CS_IO_TIMEOUT	An I/O submitted has timed out
CS_LENGTH_RETURNED	The function requested the required length and the function returned that length

Table 8: Status Value Definitions

CS_LOAD_ERROR	The specified download was not able to be initialized
CS_MEMORY_IN_USE	The requested memory was still in use
CS_NO_MATCHING_DEVICE	The indirectly specified device does not exist (e.g., an input other than device name in Path parameter may be specified as a filesystem based name such as a filename path, directory name, volume name etc that gets resolved internally to the actual device name)
CS_NO_PERMISSIONS	There were insufficient permissions to proceed with request
CS_NOT_DONE	The request was not done
CS_NOT_ENOUGH_MEMORY	There was not enough memory to satisfy the request
CS_NOTHING_QUEUED	No queued requests to poll
CS_OUT_OF_RESOURCES	The system was out of resources to satisfy the request
CS_QUEUED	The request was successfully queued
CS_SUCCESS	The action was completed with success
CS_UNKNOWN_MEMORY	The memory referenced was unknown
CS_UNSUPPORTED	The request is not supported

Table 8: Status Value Definitions

CS_UNSUPPORTED_INDEX	The specified hardware index is not supported for this download
CS_UNSUPPORTED_TYPE	The specified download type is not supported

6.3.4 Notification options

The following definitions specify the fixed defined values that may be specified as one or more notification options as input to the `csRegisterNotify()` function. The same values shall be provided to the notification callback, if invoked. See section 6.3.5.2.19 for the details on values.

Table 9: Notification Value Definitions

Status Value Definition	Description
CS_NOTIFY_SYSTEM_ERROR	A system error has occurred
CS_NOTIFY_CSE_UNRESPONSIVE	The specified CSE is not responding normally and may be unusable
CS_NOTIFY_CSEE_UNRESPONSIVE	The specified CSEE is not responding normally and may be unusable
CS_NOTIFY_CSF_UNRESPONSIVE	The specified CSF is not responding normally and may be unusable
CS_NOTIFY_CSE_RESET	A CSE resource was reset
CS_NOTIFY_CSEE_RESET	A CSEE resource was reset
CS_NOTIFY_CSx_RESET	The CSx was reset
CS_NOTIFY_CSx_ADDED	A new CSx is available
CS_NOTIFY_CSx_REMOVED	A CSx is not available

Table 9: Notification Value Definitions

CS_NOTIFY_CSF_ADDED	A new CSF was loaded
CS_NOTIFY_CSF_REMOVED	A CSF was unloaded
CS_NOTIFY_RESOURCE_WARNING	The CSx is running out of resources
CS_NOTIFY_DOWNLOAD_INFO	Additional information is available for downloaded CSF
CS_NOTIFY_CONFIG_INFO	Additional information is available for downloaded configuration

6.3.5 Data structures

6.3.5.1 Definitions

6.3.5.2 Enumerations

The enumerations in this section are used in function parameters and data structures.

6.3.5.2.1 *CS_RESOURCE_TYPE*

```
typedef enum {
    CS_CSx_TYPE               = 1,
    CS_CSE_TYPE               = 2,
    CS_CSEE_TYPE              = 3,
    CS_FDM_TYPE               = 4,
    CS_CSF_TYPE               = 5,
    CS_VENDOR_SPECIFIC_TYPE   = 6
} CS_RESOURCE_TYPE;
```

6.3.5.2.2 *CS_CSEE_RESOURCE_TYPE*

```
typedef enum {
    CS_CSEE_UNDEFINED           = 0,
    CS_CSEE_FPGA                = 1,
    CS_CSEE_BPF                 = 2,
    CS_CSEE_CONTAINER           = 3,
    CS_CSEE_OPERATING_SYSTEM    = 4,
    CS_CSEE_VENDOR_SPECIFIC     = 65535
} CS_CSEE_RESOURCE_TYPE;
```

6.3.5.2.3 *CS_CSF_RESOURCE_TYPE*

```
typedef enum {
    CS_CSF_UNDEFINED                = 0,
    CS_CSF_FPGA_BITSTREAM           = 1,
    CS_CSF_BPF_PROGRAM              = 2,
    CS_CSF_CONTAINER_IMAGE          = 3,
    CS_CSF_OPERATING_SYSTEM_IMAGE   = 4,
    CS_CSF_VENDOR_SPECIFIC          = 65535
} CS_CSF_RESOURCE_TYPE;
```

6.3.5.2.4 *CS_RESOURCE_SUBTYPE*

The following enum defines CSEE and CSF resource subtypes that may be used with the `csCSEEDownload()` function and the `csCSFDownload()` function.

```
typedef enum {
    CS_SUBTYPE_UNDEFINED              = 0,
    CS_SUBTYPE_FPGA_AMD               = 1,
    CS_SUBTYPE_FPGA_INTEL             = 2,
    CS_SUBTYPE_CONTAINER_DOCKER       = 3,
    CS_SUBTYPE_CONTAINER_KUBERNETES   = 4,
    CS_SUBTYPE_OS_LINUX               = 5,
    CS_SUBTYPE_OS_VMWARE              = 6,
    CS_SUBTYPE_VENDOR_SPECIFIC        = 65535
} CS_RESOURCE_SUBTYPE;
```

6.3.5.2.5 *CS_STATE*

```
typedef enum {
    CS_INACTIVE_STATE           = 0,
    CS_ACTIVE_STATE             = 1,
} CS_STATE;
```

6.3.5.2.6 *CS_CONFIG_TYPE*

```
typedef enum {
    CS_CSEE_ACTIVATE            = 1,
    CS_CSF_ACTIVATE             = 2,
    CS_FDM_ACTIVATE             = 3,
    CS_VENDOR_SPECIFIC          = 4
} CS_CONFIG_TYPE;
```

6.3.5.2.7 ***CS_FDM_FLAG_TYPE***

CS_FDM_FLAG_TYPE specifies options to csMemFlags parameter for the `csAllocMem()` function and the `csFreeMem()` function.

```
typedef enum {
    CS_FDM_CLEAR           = 1,          // clears AFDM to all zeroes
    CS_FDM_FILL            = 2           // fill AFDM with specified value
} CS_FDM_FLAG_TYPE;
```

More than one option may be specified for CS_FDM_FLAG_TYPE; therefore, this is defined as a bitmask.

6.3.5.2.8 ***CS_MEM_COPY_TYPE***

```
typedef enum {
    CS_COPY_TO_DEVICE      = 1,
    CS_COPY_FROM_DEVICE    = 2,
    CS_COPY_WITHIN_DEVICE  = 3
} CS_MEM_COPY_TYPE;
```

6.3.5.2.9 ***CS_STORAGE_REQ_MODE***

```
typedef enum {
     CS_STORAGE_BLOCK_IO   = 1,
     CS_STORAGE_FILE_IO    = 2
} CS_STORAGE_REQ_MODE;
```

6.3.5.2.10 ***CS_STORAGE_IO_TYPE***

```
typedef enum {
     CS_STORAGE_LOAD_TYPE         = 1,
     CS_STORAGE_STORE_TYPE        = 2
} CS_STORAGE_IO_TYPE;
```

6.3.5.2.11 ***CS_COMPUTE_ARG_TYPE***

This `enum` defines the CSF argument types.

```
typedef enum {
    CS_AFDM_TYPE           = 1,
    CS_8BIT_VALUE_TYPE     = 2,
    CS_16BIT_VALUE_TYPE    = 3,
    CS_32BIT_VALUE_TYPE    = 4,
    CS_64BIT_VALUE_TYPE    = 5,
```

```
    CS_128BIT_VALUE_TYPE   = 6,
    CS_DESCRIPTOR_TYPE            = 7
} CS_COMPUTE_ARG_TYPE;
```

6.3.5.2.12 *CS_BATCH_MODE*

This `enum` enumerated the possible batch modes as follows:

```
typedef enum {
     CS_BATCH_SERIAL       = 1,
     CS_BATCH_PARALLEL     = 2,
     CS_BATCH_HYBRID       = 3
} CS_BATCH_MODE;
```

6.3.5.2.13 *CS_BATCH_REQ_TYPE*

```
typedef enum {
     CS_COPY_AFDM          = 1,
     CS_STORAGE_IO         = 2,
     CS_QUEUE_COMPUTE      = 3
} CS_BATCH_REQ_TYPE;
```

6.3.5.2.14 *CS_BATCH_CONFIG_TYPE*

```
typedef enum {
     CS_BATCH_ADD          = 1,
     CS_BATCH_DELETE       = 2,
     CS_BATCH_RECONFIG     = 3,
     CS_BATCH_JOIN         = 4,
     CS_BATCH_SPLIT        = 5
} CS_BATCH_CONFIG_TYPE;
```

6.3.5.2.15 *CS_STAT_TYPE*

This data type defines various statistics that are able to be queried from a CSx.

```
typedef enum {
     CS_STAT_CSE_USAGE            = 1,   // query to provide CSE runtime statistics
     CS_STAT_CSx_MEM_USAGE        = 2,   // query CSx memory usage
     CS_STAT_CSF                  = 3    // query statistics on a specific function
} CS_STAT_TYPE;
```

6.3.5.2.16 *CS_LIBRARY_SUPPORT*

```
typedef enum {
     CS_FILE_SYSTEMS_SUPPORTED   = 1,
```

```
     CS_RESERVED                 = 2
} CS_LIBRARY_SUPPORT;
```

6.3.5.2.17 ***CS_PLUGIN_TYPE***

```
typedef enum {
     CS_PLUGIN_COMPUTE           = 1,
     CS_PLUGIN_NVME              = 2,
     CS_PLUGIN_FILE_SYSTEM       = 4,
     CS_PLUGIN_CUSTOM            = 8
} CS_PLUGIN_TYPE;
```

A plugin may be more than one type; therefore, this is defined as a bitmask.

6.3.5.2.18 ***CS_STATUS***

This data type defines status values that may be returned by the interface functions. See Table 8 for details on each status.

```
typedef enum CS_STATUS {
     CS_SUCCESS                  = 0,
     CS_COULD_NOT_MAP_MEMORY     = 1,
     CS_DEVICE_ERROR             = 2,
     CS_DEVICE_NOT_AVAILABLE     = 3,
     CS_DEVICE_NOT_READY         = 4,
     CS_DEVICE_NOT_PRESENT       = 5,
     CS_INVALID_DEVICE_NAME      = 6,
     CS_INVALID_PATH             = 7,
     CS_ENTITY_NOT_ON_DEVICE     = 8,
     CS_NO_MATCHING_DEVICE       = 9,
     CS_ERROR_IN_EXECUTION       = 10,
     CS_FATAL_ERROR              = 11,
     CS_HANDLE_IN_USE            = 12,
     CS_INVALID_HANDLE           = 13,
     CS_INVALID_ARG              = 14,
     CS_INVALID_ID               = 15,
     CS_INVALID_LENGTH           = 16,
     CS_INVALID_OPTION           = 17,
     CS_INVALID_CSF_ID           = 18,
     CS_INVALID_CSF_NAME         = 19,
     CS_INVALID_FDM_ID           = 20,
     CS_INVALID_GLOBAL_ID        = 21,
     CS_IO_TIMEOUT               = 22,
     CS_LOAD_ERROR               = 23,
     CS_MEMORY_IN_USE            = 24,
     CS_NO_PERMISSIONS           = 25,
     CS_NOT_DONE                 = 26,
     CS_NOT_ENOUGH_MEMORY        = 27,
     CS_OUT_OF_RESOURCES         = 28,
     CS_QUEUED                   = 29,
     CS_NOTHING_QUEUED           = 30,
     CS_COULD_NOT_UNMAP_MEMORY   = 31,
     CS_UNKNOWN_MEMORY           = 32,
```

```
    CS_UNSUPPORTED              = 33,
    CS_UNSUPPORTED_TYPE         = 34,
    CS_UNSUPPORTED_INDEX        = 35
} CS_STATUS;
```

6.3.5.2.19 *CS_NOTIFY_TYPE*

This data type defines notification types that may be specified as one or more values as input to the `csRegisterNotify()` function. The same definitions shall be provided to the notification callback, if invoked. See Table 9 for details on each notification type.

```
typedef enum {
    CS_NOTIFY_SYSTEM_ERROR      = 1 << 0,
    CS_NOTIFY_CSE_UNRESPONSIVE  = 1 << 1,
    CS_NOTIFY_CSEE_UNRESPONSIVE = 1 << 2,
    CS_NOTIFY_CSF_UNRESPONSIVE  = 1 << 3,
    CS_NOTIFY_CSE_RESET         = 1 << 4,
    CS_NOTIFY_CSEE_RESET        = 1 << 5,
    CS_NOTIFY_CSx_RESET         = 1 << 6,
    CS_NOTIFY_CSx_ADDED         = 1 << 7,
    CS_NOTIFY_CSx_REMOVED       = 1 << 8,
    CS_NOTIFY_CSF_ADDED         = 1 << 9,
    CS_NOTIFY_CSF_REMOVED       = 1 << 10,
    CS_NOTIFY_RESOURCE_WARNING  = 1 << 11,
    CS_NOTIFY_DOWNLOAD_INFO     = 1 << 12,
    CS_NOTIFY_CONFIG_INFO       = 1 << 13
} CS_NOTIFY_TYPE;
```

6.3.5.3 Structures

The structures in this section are used in function parameters and within other data structures.

6.3.5.3.1 ***Properties Data Structures***

The following data structures are used for discovery of all resources for a CSx. The data structure `CSProperties` is queried using the `csQueryDeviceProperties()` function and provides the properties for all compute resources of a CSx. The structure contains sub-structures that are required to be queried individually using the `CS_RESOURCE_TYPE` enumerator. The discoverable sub-structures include `CSxProperties`, `CSEProperties`, `CSEEProperties`, `FDMProperties`, and `CSFProperties`.

The sub-structure `CSxProperties` provides information pertaining to the CSx. The `BatchRequestsSupported` field specifies if this CSx supports batch requests in hardware.

The sub-structure `CSEProperties` provides information on all CSEs, where each one is described by the sub-structure `CSEInfo`. The field `CSETypeToken` is a device specified entry that uniquely distinguishes between different CSE types. The MaxRequestsPerBatch field denotes the maximum number of requests that may be batched together in a batch request using the csQueueBatchRequest() function. The MaxCSFParametersAllowed field denotes the maximum parameters supported for a given CSF by the CSE. A function cannot exceed this number and will be rejected if it does by the queueing function. Each CSE is further identified by the sub-structure ComputeResource that provides individual details on the CSE.

The sub-structure CSEEProperties provides details on the execution environment and is associated to the CSE by the CSETypeToken field. Each CSEE represented by CSEEInfo sub-structure may describe any CSFs that are built-in or preloaded in the CSx by the field NumBuiltinCSFs. The NumActivatedCSFs field denotes the total number of CSFs available for execution.

The sub-structure FDMProperties describes all FDMs available on the CSx. Each FDM is described by sub-structure FDMInfo. The DeviceManaged field when set to 1 identifies that the CSx manages FDM for allocations and deallocations and determines how the memory is managed. If this field is set to zero, it means that the host manages this resource. The HostVisible field when set to 1 denotes that FDM is available as a physical resource in the system's address space and may be mapped into a host's virtual address. If set to zero, FDM is not visible and requires specific functions to operate on it. The ClearContentsSupported and InvalidateContentsSupported fields when set to 1 specify if the FDM supports this feature in hardware. The ConfigSupported field specifies if the FDM may be configured. The NumCSEs field denotes the CSEs that have access to the FDM while CSEList field is a pointer that provides the details on the individual CSEs described in CSEAccess data structure.

The sub-structure CSFProperties describes all the CSFs that are available on the CSx. Each CSF is described by the sub-structure CSFInfo that describes its association to a CSE by the CSETypeToken field. Additional fields such as CSFId uniquely identify the CSF and BuiltIn verify if the CSF is built-in/preloaded by the vendor. A CSF is required to be activated to be able to run on a CSE and its activation state and associations is further described by the sub-structure CSFInstance. A CSF may be executed on more than one CSE if allowed by that engine type.

6.3.5.3.1.1 CsProperties

```
typedef union {
     CSxProperties CSxDetails;          // details on CSx
     CSEProperties CSEDetails[1];       // details on all CSEs
     CSEEProperties CSEEDetails[1];     // details on all CSEEs
     FDMProperties FDMDetails[1];       // details on all FDMs
```

```
     CSFProperties CSFDetails[1];        // details on all CSFs
     CSVendorSpecific VSDetails;         // vendor specific
} CsProperties;
```

6.3.5.3.1.2 CSxProperties

```
typedef struct {
     u16 HwVersion;         // specifies the hardware version of this CSx
     u16 SwVersion;         // specifies the software version that runs on this CSx
     u32 VendorId;          // specifies the vendor identifier of this CSx
     u32 DeviceId;          // specifies the device identifier of this CSx
     char FriendlyName[32]; // an identifiable string for this CSx
     struct {
             u64 BatchRequestsSupported : 1;// CSx supports batch requests in hardware
             u64 Reserved : 63;
     } Flags;
} CSxProperties;
```

6.3.5.3.1.3 ComputeResource

```
typedef struct {
     u16 HwVersion;
     u16 SwVersion;
     char Name[32];         // an identifiable string for this CR
     u32 ERId;              // Engine Resource Identifier for this ComputeResource
     CRProperties *Features; // additional features like perf, security etc [TBD]
} ComputeResource;
```

6.3.5.3.1.4 CSEInfo

```
typedef struct {
     u16 CSETypeToken;             // device provided token to differentiate between its
                                   // CSE types
     u8 RelativePerformance;       // values [1-10]; higher is better; 0 is not defined
     u8 RelativePower;             // values [1-10]; lower is better; 0 is not defined
     u32 MaxRequestsPerBatch;      // maximum number of requests supported per
                                   // batch request
     u32 MaxCSFParametersAllowed;// maximum number of parameters supported
     u32 CSEId;                    // CSE identifier unique to this CSx
     u16 MaxCSEEs;                 // maximum number of CSEEs for this CSE
     u16 NumActivatedCSEEs;        // number of activated CSEEs
     u16 NumAvailableCRs;          // number of CRs not allocated
     u16 NumCRs;                   // total CRs in list
     ComputeResource *CRs;         // a pointer to a list of CRs for this CSE Type
} CSEInfo;
```

6.3.5.3.1.5 CSEProperties

```
typedef struct {
     u16 NumCSEs;                  // number of CSEs in array
     CSEInfo CSE[1];               // a array of CSEs
} CSEProperties;
```

6.3.5.3.1.6 CSEEInstance

```
typedef struct {
     CS_STATE State;               // current activation state
     u32 CSEId;                    // CSE identifier unique to this CSx
     u32 EEIId;                    // Execution Environment Instance Identifier
} CSEEInstance;
```

6.3.5.3.1.7 CSEEInfo

```
typedef struct {
     CS_CSEE_RESOURCE_TYPE Type;        // the type of CSEE
     u16 SwVersion;
     char Name[32];               // an identifiable string for this CSEE
     u16 CSETypeToken;            // device provided token to differentiate between
                                  // its CSE types
     u16 NumBuiltinCSFs;          // number of available vendor preloaded CSFs
     u32 CSEEId;                  // unique CSEE identifier
     u16 MaxCSFs;                 // maximum number of CSFs for this CSEE
     u16 NumActivatedCSFs;        // number of activated CSFs
     u16 NumEEIs;                 // number of activated CSEE instances
     CSEEInstance *EEInstances;   // a pointer to a list of activated CSEE instances
} CSEEInfo;
```

6.3.5.3.1.8 CSEEProperties

```
typedef struct {
     u16 NumCSEEs;                // number of CSEEs in array
     CSEEInfo CSEE[1];            // an array of CSEEs
} CSEEProperties;
```

6.3.5.3.1.9 FDMFlags

```
typedef struct {
      u64 DeviceManaged: 1;              // FDM allocations managed by device
      u64 HostVisible: 1;                // FDM may be mapped to host address space
      u64 ClearContentsSupported: 1;     // supports clearing FDM with zeros
      u64 InvalidateContentsSupported: 1;// supports invalidating FDM with non-zeros
      u64 ConfigSupported: 1;            // supports configuration
      u64 Reserved: 59;
} FDMFlags;
```

6.3.5.3.1.10 CSEAccess

```
typedef struct {
     u32 CSEId;                   // CSE identifier unique to this CSx
     u8 RelativePerformance;      // values [1-10]; higher is better; 0 is not defined
     u8 RelativePower;            // values [1-10]; lower is better; 0 is not defined
} CSEAccess;
```

6.3.5.3.1.11 FDMInfo

```
typedef struct {
     u32 FDMId;                   // unique FDM identifier
     u64 FDMSize;                 // size of FDM in bytes
     FDMFlags Flags;              // FDM Settings
     u16 NumCSEs;                 // total CSEs in list
     CSEAccess *CSEList;          // a pointer to a list of CSEs having access
} FDMInfo;
```

6.3.5.3.1.12 FDMProperties

```
typedef struct {
     u16 NumFDMs;                 // number of FDMs in array
```

```
     FDMInfo FDM[1];            // an array of FDMs (as applicable)
} FDMProperties;
```

6.3.5.3.1.13 CSFInstance

```
typedef struct {
     CS_STATE State;            // current activation state
     u32 EEIId;                 // paired CSEE instance identifier
     u32 FIId;                  // unique CSF Instance identifier
     u16 NumCRs;                // number of CRs in CRList
     u32 *ERList;               // pointer to a list of CR identifiers on which a CSF
                                // instance is activated
} CSFInstance;
```

6.3.5.3.1.14 CSFInfo

```
typedef struct {
     u64 GlobalId;              // global unique identifier assigned to the CSF
     char UniqueName[32];       // an identifiable string for this CSF, if available
     u16 CSETypeToken;          // device provided token to differentiate between its
                                // CSE types
     u32 CSFId;                 // unique CSF identifier
     u8 Builtin: 1;             // preloaded by vendor
     u8 Reserved: 7;
     u16 NumFIs;                // number of associated instances for this CSF
     CSFInstance *FInstances;   // pointer to list of CSF instances with CSEE & CR
     details
} CSFInfo;
```

6.3.5.3.1.15 CSFProperties

```
typedef struct {
     u16 NumCSFs;               // number of CSFs in CSFInfo array
     CSFInfo CSF[1];            // an array of CSFs
} CSFProperties;
```

6.3.5.3.1.16 CSVendorSpecific

```
typedef struct {
     void *VSData;
} CSVendorSpecific;
```

6.3.5.3.2 *Configuration Data Structures*

The data structure `csConfigInfo` is provided as input to configure a CSx using the `csConfig()` function. On success, the data structure `CsConfigData` provides the results of the requested configuration. Configurations are selected using the `CS_CONFIG_TYPE` enumerator.

The CSEE resource may be configured with a CSE resource that matches its resource type by the `CSETypeToken` field in their respective data structures. Configuring these resources together is described using the `CSEEConfig` data structure. On successful configuration, the a unique Execution Environment Instance Identifier (EEIId) along with the activation state set is returned as a result in sub-data structure `CsActivationInfo`. The EEIId identifier is primarily used for activating a CSF.

Similarly, the CSF resource may be configured with a valid EEIId and one or more compute resources (CRs). The configuration request is valid for the same `CSETypeToken` types i.e., an activation may only be performed on the same `CSETypeToken` types. On successful configuration, the sub-data structure `CsActivationInfo` is populated with the unique Functional Instance Identifier (FIId) and the resultant activated state. The FIId is a unique instance of a CSF. There can be multiple activated FIIds for a single CSF. The maximum number of CSFs that may be activated is dependent on the CSE. Only activated CSFs are visible when queried using the `csGetCSFId()` function and the `csQueryCSFList()` function. Only activated CSF instances are used in execution using the `csQueueComputeRequest()` function or the `csQueueBatchRequest()` function.

The FDM resource may be configured using the `FDMConfig` sub-structure. The specified FDMId may be configured to the specified `State` field only if the FDM supports the state as specified in `FDMInfo`. If successful, the applied state is reflected in the `State` field of `CsActivationInfo`.

6.3.5.3.2.1 CsConfigInfo

The data structure `CsConfigInfo` is defined as follows:

```
typedef struct {
	CS_CONFIG_TYPE Type;
	union {
		CSEEConfig CSEEActivateInfo;	// configuration details for CSEE
		CSFConfig CSFActivateInfo;	// configuration details for CSF
		FDMConfig FDMActivateInfo;	// configuration details for FDM
		CSVendorConfig VSInfo;	// vendor specific
	} u;
} CsConfigInfo;
```

6.3.5.3.2.2 CsConfigData

```
typedef union {
	CsActivationInfo Data;
	void *VSData;
} CsConfigData;
```

6.3.5.3.2.3 CSEEConfig

```
typedef struct {
	CS_STATE State;		// requested activation state
	u32 CSEEId;		// unique CSEE Identifier
	u32 CSEId;		// CSE identifier unique to CSx
```

```
} CSEEConfig;
```

6.3.5.3.2.4 CSFConfig

```
typedef struct {
     CS_STATE State;            // requested activation state
     u32 CSFId;                 // unique CSF identifier
     u32 EEIId;                 // Execution Environment Instance Identifier
     U16 NumCRs;                // number of CRs in array
     u32 CRArray[1];            // an array of one or more Compute Resources
                                // (ERIds see 6.3.5.3.1.3)
} CSFConfig;
```

6.3.5.3.2.5 FDMConfig

```
typedef struct {
     CS_STATE State;            // requested configuration state
     u32 FDMId;                 // requested FDM configuration by identifier
} FDMConfig;
```

6.3.5.3.2.6 CSVendorConfig

```
typedef struct {
     void *VSData;
} CSVendorConfig;
```

6.3.5.3.2.7 CsActivationInfo

```
typedef struct {
     CS_STATE State;            // current activation state
     u32 ID;                    // resource specific unique Identifier
} CsActivationInfo;
```

6.3.5.3.3 ***Memory Data Structures***

The memory data structures provide the definitions on how memory is organized and for its access usage with the necessary functions. Memory is represented by its memory handle and is required to be allocated using the `csAllocMem()` function prior to usage.

6.3.5.3.3.1 CsMemFlags

The data structure `CsMemFlags` provides inputs to the `csAllocMem()` function specifying:

- the FDM (i.e., in the FDMId field) from which to allocate memory; and

- flags that specify how the memory is to be initialized.

The value for `FDMId` is queried by CSF using the the `csGetCSFId()` function.

```
typedef struct {
     u32 FDMId;                     // refer to the csGetCSFId()function for
     details
     CS_FDM_FLAG_TYPE Flags;        // see 6.3.5.2.7
     u32 FillValue;                 // only valid when fill flag is specified
} CsMemFlags;
```

6.3.5.3.3.2 CsDevAFDM

The data structure `CsDevAFDM` defines how memory may be used and defines a previously allocated memory handle and an offset denoted by `ByteOffset` to reference within that memory.

```
typedef struct {
     CS_MEM_HANDLE MemHandle;       // an opaque memory handle for AFDM
     unsigned long ByteOffset;      // denotes the offset with AFDM
} CsDevAFDM;
```

6.3.5.3.3.3 CsCopyMemRequest

The structure `CsCopyMemRequest` describes the memory copy request between the host memory and the AFDM. A `CsCopyMemRequest` is able to describe a copy from host memory to the AFDM or from the AFDM to host memory based on the `Type` field.

```
typedef struct {
     CS_MEM_COPY_TYPE Type;         // see 6.3.5.2.8
      union {
         void *HostVAddress;        // defines host memory if specified in Type
         CsDevAFDM SrcDevMem;       // defines the source device memory for copy
                                    // between device memories
      } u;
     CsDevAFDM DevMem;              // see 6.3.5.3.3.1
     int Bytes;
} CsCopyMemRequest;
```

6.3.5.3.4 ***Storage Data Structures***

The structure `CsStorageRequest` describes the storage I/O request between the storage device and the CSF. Storage I/O is a block or file request and utilizes the `Mode` field to select block I/O or file I/O. The `Type` field describes the direction of data flow from storage device.

Block requests describe details such as the namespace to operate on, the LBA, and number of blocks to transfer. Multiple LBA block ranges may be specified in the same request. They also describe the AFDM that the transfer occurs to or from. The `StorageIndex` field specifies the drive to target the request to in a CSA and is reserved for other CSx types. See `CsDevAFDM` data structure for details on the `DevMem` field as specified in section 6.3.5.3.3.1.

For file requests, the `CsFileIo` structure describes the file request to perform with details of the file handle, offset within the file, bytes to read/write, and device memory buffer details. File based requests will be satisfied for the default file system(s) for that OS. A specific file system support should be first queried before making a file-based request. The handle is required to refer to a valid open file with the required set of access rights to satisfy the intent of the request. File offset and bytes requested are required to adhere to the storage drives block requirements. For file write based requests, the function will synchronize on writing to that portion of the file with the filesystem and reserve space in advance, if required. File based requests are translated internally to a storage I/O request. See section 6.13.1 for more information on file system support.

6.3.5.3.4.1 CsStorageRequest

The data structure `CsStorageRequest` is defined as follows:

```
typedef struct {
     CS_STORAGE_REQ_MODE Mode;           // see 6.3.5.2.9
     CS_DEV_HANDLE DevHandle;            // the CSx handle
     union {
      CsBlockIo BlockIo;                 // see 6.3.5.3.4.1
      CsFileIo FileIo;                   // see 6.3.5.3.4.4
     } u;
} CsStorageRequest;
```

6.3.5.3.4.2 CSBlockRange

The data structure `CsBlockRange` is defined as follows:

```
typedef struct {
     u32 NamespaceId;                    // represents a LUN or namespace
     u64 StartLba;                       // the starting LBA for this range
     u32 NumBlocks;                      // total number of blocks for this range
} CsBlockRange;
```

6.3.5.3.4.3 CsBlockIo

The data structure `CsBlockIo` is defined as follows:

```
typedef struct {
     CS_STORAGE_IO_TYPE Type;            // see 6.3.5.2.10
     u32 StorageIndex;                   // denotes the index in a CSA, zero otherwise
```

```
     CsDevAFDM DevMem;                  // see 6.3.5.3.3.1
     int NumRanges;                     // number of LBA block ranges
     CsBlockRange Range[1];             // An array of LBA block ranges
} CsBlockIo;
```

6.3.5.3.4.4 CsFileIo

The data structure `CsFileIo` is defined as follows:

```
typedef struct {
     CS_STORAGE_IO_TYPE Type;           // see 6.3.5.2.10
     void *FileHandle;
     u64 Offset;
     u32 Bytes;
     CsDevAFDM DevMem;                  // see 6.3.5.3.3.1
} CsFileIo;
```

6.3.5.3.5 ***Compute Data Structures***

Compute requests are described using the `CsComputeRequest` data structure. The `CSFId` data field holds the identifier of the CSF that has to be executed. The `NumArgs` field describes the total number of arguments passed down to the CSF while `Args` describes the first argument. `Args` may be described in an array where, the total count in the array is described by `NumArgs` field.

The `Args` field is described by the `CsComputeArg` data structure. The `Type` field denotes the argument type while the details are one of the types in the union.

6.3.5.3.5.1 CsComputeRequest

The structure `CsComputeRequest` is an input to schedule and run a CSF. The arguments are function dependent.

```
typedef struct {
     CS_CSF_ID CSFId;            // A unique identifier for a Computational Storage
                                 // Function within a CSx see 6.3.7
     int NumArgs;                // set to total arguments to CSF
     CsComputeArg Args[1];       // see 6.3.5.3.5
                                 // allocate enough space past this for multiple
} CsComputeRequest;              // arguments
```

6.3.5.3.5.2 CsComputeArg

The structure `CsComputeArg` describes an individual argument to a CSF. A handle references AFDM while the values refer to scalar inputs to the CSF.

```
typedef struct {
```

```
    CS_COMPUTE_ARG_TYPE Type;
    union {
     CsDevAFDM DevMem;          // see 6.3.5.3.3.1
     u64 Value64;
     u32 Value32;
     u16 Value16;
     u8 Value8;
    } u;
} CsComputeArg;
```

6.3.5.3.6 ***Batch Data Structures***

Batch requests help optimize the total number of function requests by combining multiple requests into one batch. Batch requests also help execute repeatable tasks. Batch request setup is defined in detail under section 5.2.

Each request in a batch is described by the `CsBatchRequest` data structure. The `ReqType` data field describes the type of batch request.

6.3.5.3.6.1 CsBatchRequest

The data structure `CsBatchRequest` is defined as follows:

```
typedef struct {
    CS_BATCH_REQ_TYPE ReqType;          // see 6.3.5.2.13
    union {
     CsCopyMemRequest CopyMem;          // see 6.3.5.3.3.3
     CsStorageRequest StorageIo;        // see 6.3.5.3.4.1
     CsComputeRequest Compute;          // see 6.3.5.3.5.1
    } u;
} CsBatchRequest;
```

6.3.5.3.7 ***Statistics Data Structures***

CSx statistics for specific resources may be queried using the `csQueryDeviceStatistics()` function.

The `Stats` parameter defined as `CsStatsInfo` structure is used to query a specific statistic type as provided by the `Type` input parameter. The optional `Identifier` parameter may be provided if `Type` requires a valid identifier. For example, the `CSFId` may be provided as the `Identifier` parameter to query the particular CSF's usage statistics as defined in `CSFUsage` data structure.

6.3.5.3.7.1 CsStatsInfo

The data structure `CsStatsInfo` is defined as follows:

```
typedef union {
    CSEUsage CSEDetails;
    CSxMemory MemoryDetails;     // see 6.3.5.3.7.3
    CSFUsage CSFDetails;         // see 6.3.5.3.7.4
```

```
} CsStatsInfo;
```

6.3.5.3.7.2 CSEUsage

`CSEUsage` provides the following details when queried for a particular CSE. The counters reflect numbers since the device was last reset.

```
typedef struct {
     u32 PowerOnMins;
     u32 IdleTimeMins;
     u64 TotalFunctionExecutions;// total number of executions performed by CSE
} CSEUsage;
```

6.3.5.3.7.3 CSxMemory

CSxMemory defines device memory usage.

All counters are represented in bytes if not specified.

```
typedef struct {
     u64 TotalAllocatedFDM;            // total FDM in bytes that have been allocated
     u64 LargestBlockAvailableFDM;     // largest amount of FDM that may be allocated
     u64 AverageAllocatedSizeFDM;      // average size of FDM allocations in bytes
     u64 TotalFreeFDM;                 // total FDM memory that is not in use
     u64 TotalAllocationsFDM;          // count of total number of FDM allocations
     u64 TotalDeAllocationsFDM;        // count of total number of FDM deallocations
     u64 TotalFDMtoHostinMB;           // total FDM transferred to host memory in
                                       // megabytes
     u64 TotalHosttoFDMinMB;           // total host memory transferred to FDM in
                                       // megabytes
     u64 TotalFDMtoStorageinMB;  // total FDM transferred to storage in megabytes
     u64 TotalStoragetoFDMinMB;  // total storage transferred to FDM in megabytes
} CSxMemory;
```

6.3.5.3.7.4 CSFUsage

CSFUsage defines per function statistics since the function was loaded. The counters are cleared when the function gets unloaded. The specific function is chosen as input with the `Identifier` parameter.

```
typedef struct {
     u64 TotalUptimeSeconds;     // total utilized time by CSF in seconds
     u64 TotalExecutions;        // number of executions performed
     u64 ShortestTimeUsecs;      // the shortest time the CSF ran in microseconds
     u64 LongestTimeUsecs;       // the longest time the CSF ran in microseconds
     u64 AverageTimeUsecs;       // the average runtime in microseconds
} CSFUsage;
```

6.3.5.3.8 ***FDMAccess***

The data structure `FDMAccess` specifies the FDM access by `FDMId` for a given CSF and is defined as follows:

```
typedef struct {
     u32 FDMId;                  // Unique FDM identifier that is used to allocate FDM
     u8 RelativePerformance;     // values [1-10]; higher is better; 0 is not defined
     u8 RelativePower;           // values [1-10]; lower is better; 0 is not defined
     FDMFlags Flags;             // FDM settings
} FDMAccess;
```

6.3.5.3.9 *CSFUniqueId*

```
typedef struct {
     char UniqueName[32];        // an identifiable string for CSF if available,
                                 // NULL otherwise
     u64 GlobalId;               // global unique identifier for CSF if available,
                                 // zeroes otherwise
} CSFUniqueId;
```

6.3.5.3.10 *CSFIdInfo*

The data structure `CSFIdInfo` is returned on a successful query from the `csGetCSFId()` function and is defined as follows:

```
typedef struct {
     CS_CSF_ID CSFId;            // unique CSF Identifier used to schedule compute
                                 // work
     u8 RelativePerformance;     // values [1-10]; higher is better; 0 is not defined
     u8 RelativePower;           // values [1-10]; lower is better; 0 is not defined
     u8 Count;                   // number of instances of this CSF available
     u8 NumFDMs;                 // number of FDMs accessible by the CSF
     FDMAccess *FDMList;         // list of accessible FDMs
} CSFIdInfo;
```

6.3.5.3.11 *CsCommonDownloadInfo*

This is a sub-structure of `CsCSFDownloadInfo` and `CsCSEEDownloadInfo` data structures. The `UniqueName` field is an identifiable string for the resource being downloaded, if available. The `Unload` field is only set if a previously downloaded resource at `Index` in the main data structure is to be unloaded. This field is set to zero otherwise. The `Length` and `DataBuffer` fields denote the length and contents of the resource being downloaded.

```
typedef struct {
     char UniqueName[32];        // an identifiable string for resource, if available
     int Unload;                 // unload previously loaded entity, zero otherwise
     int Length;                 // length in bytes of data in DataBuffer
     void *DataBuffer;           // download data for CS resource
} CsCommonDownloadInfo;
```

6.3.5.3.12 *CsCSEEDownloadInfo*

The data structure `CsCSEEDownloadInfo` contains download information for a CSEE resource based on the enumerated `Type` field. The `Type` field is required to be set to one of CS_CSEE_RESOURCE_TYPE definitions (see 6.3.5.2.2). The `SubType` field provides additional information on the download for the chosen resource (see 6.3.5.2.4. The `CSEId` field refers to a CSE identifier the CSEE is being downloaded to while the `Index` field is a hardware specific index for the CSE chosen.

If `Unload` field in `common` is set, then only the `CSEId` and `Index` fields are valid.

```
typedef struct {
     CS_CSEE_RESOURCE_TYPE Type;
     CS_RESOURCE_SUBTYPE SubType;       // value dependent on resource type
     u32 CSEId;                         // CSE identifier to download the resource to
     u32 Index;                         // A hardware-based index where the download
                                        // resides
     CsCommonDownloadInfo common;       // common fields (refer to 6.3.5.3.11)
} CsCSEEDownloadInfo;
```

6.3.5.3.13 *CsCSFDownloadInfo*

The data structure `CsCSFDownloadInfo` contains download information for a CSF resource based on the enumerated `Type` field. The `Type` field is required to be set to one of CS_CSF_RESOURCE_TYPE definitions (see 6.3.5.2.3). The `SubType` field provides additional information on the download for the chosen resource (see 6.3.5.2.4). The `GlobalId` field, if non-zero, refers to a global unique identifier for the CSF being downloaded. The `CSEEId` field refers to a CSEE identifier the CSF is being downloaded to while the `Index` field is a hardware specific index for the CSEE chosen.

If `Unload` field in `common` is set, then only the `CSEEId` and `Index` fields are valid.

```
typedef struct {
     CS_CSF_RESOURCE_TYPE Type;
     CS_RESOURCE_SUBTYPE SubType;       // value dependent on resource type
     u64 GlobalId;                      // global unique identifier assigned to
                                        // the CSF, if available
     u32 CSEEId;                        // CSEE identifier to download the resource to
     u32 Index;                         // A hardware-based index where the
                                        // download resides
     CsCommonDownloadInfo common;       // common fields (refer to 6.3.5.3.11)
} CsCSFDownloadInfo;
```

6.3.5.3.14 ***CsPluginRequest***

The data structure `CsPluginRequest` is defined as follows:

```
typedef struct {
      enum CS_PLUGIN_TYPE Type;                    // see 6.3.5.2.17
      char PluginPath[4096];                       // full path to plugin
} CsPluginRequest;
```

6.3.6 Resources

Table 10: Table of resources

Resource	Details
CS_DEV_HANDLE	The global device handle returned by `csOpenCSx`
CS_MEM_HANDLE	Denotes a device memory handle and represents memory allocated on device
CS_CSF_ID	Denotes a computational storage function for all compute offload purposes
CS_EVT_HANDLE	Denotes an event handle for asynchronous I/O
CS_BATCH_HANDLE	Denotes a batch request handle
CS_REQ_HANDLE	A handle to the outstanding request

6.3.7 Resource dependency

Figure 24 describes the resource dependency for each resource.

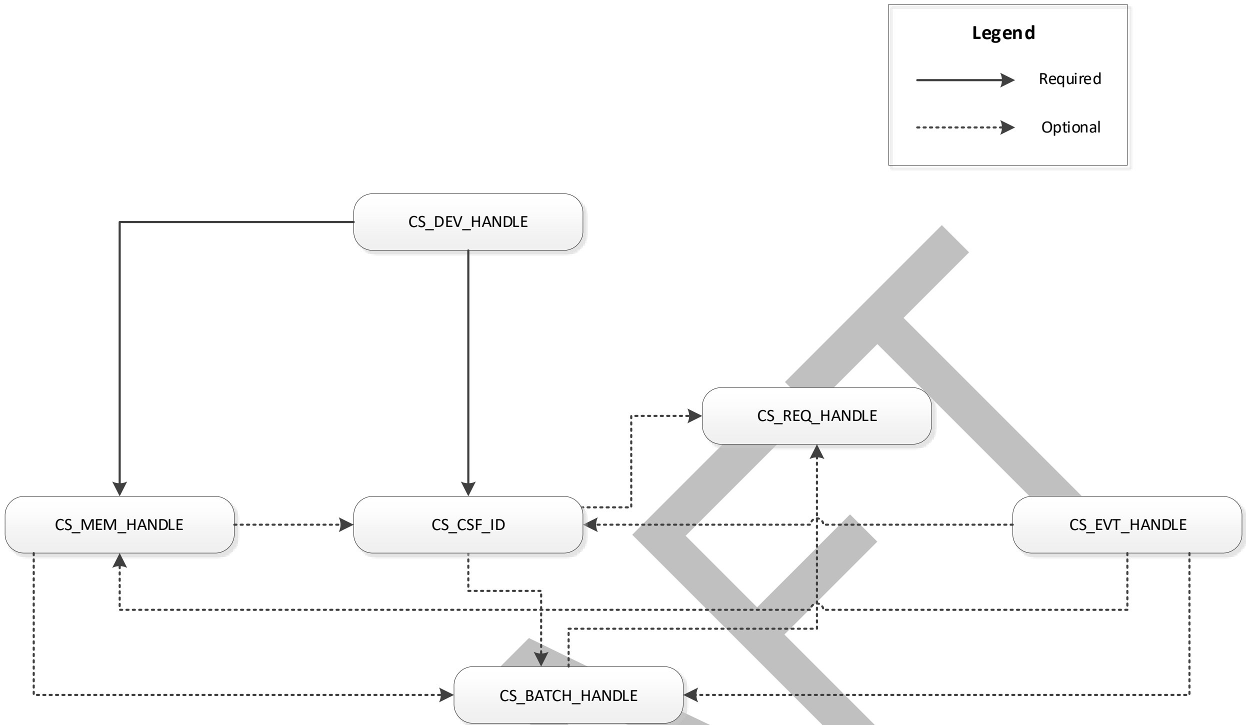

Figure 24: Resource Dependency Chart

Each resource created with the device is represented by a handle of type `CS_XXX_HANDLE` or `CS_CSF_ID` where `XXX` denotes the resource handle type. Some paths are required for the resource to be created and used while other paths may be optional.

For example, scheduling of compute offload jobs uses the `CS_CSF_ID` and may be done using synchronous or asynchronous notification mechanisms for completion. Here, `CS_EVT_HANDLE` is a notification option available that is not mandatory since an asynchronous mechanism may also be utilized with the callback option. Similarly, `CS_MEM_HANDLE` may be used by itself for device memory transfer operations.

The resource `CS_EVT_HANDLE` is a global resource while the others are allocated from the device. In a multi-device usage scenario, device specific resource handles play a key role in uniquely identifying resource by device type. The underlying implementation infrastructure will guarantee that there is no overlap between the resources and they are able to be kept unique when scaled.

6.3.8 Notification callbacks

Common callback function definition to receive notifications for various CS based events. The callback is registered through the `csRegisterNotify()` function.

```
typedef void(*csDevNotificationFn)(u32 Notification,

    void *Context, CS_STATUS Status, int Length, void *Buffer);
```

This callback is invoked with specific notification information for which the context will correspond to. If the notification is for the CSx, the context will correspond to the context specified when the CSx was opened. If the notification corresponds to a CSE, then the context will correspond to the context specified when the CSE was opened.

Common callback function definition while queuing I/O to the CSx

```
typedef void(*csQueueCallbackFn)(void *QueueContext,

    CS_STATUS Status, u64 CompValue);
```

6.4 Discovery

6.4.1 csQueryCSxList()

This function returns all of the CSxes available in the system.

6.4.1.1 Synopsis

CS_STATUS csQueryCSxList(int *Length, char *Buffer);

6.4.1.2 Parameters

IN OUT Length	A pointer to a buffer that holds or is able to hold the length of `Buffer`
OUT Buffer	Pointer to a buffer to hold a CSx list

6.4.1.3 Description

The `csQueryCSxList()` function fills `Buffer` with a comma separated list of all known CSxes identified by CSx names, if the length specified in `Length` is sufficient. This function may return zero or more CSxes as a list in `Buffer` when there are multiple CSxes devices in the system. If the length specified in `Length` is not sufficient to hold the contents returned in `Buffer`, then `Length` is populated with the required buffer size and a status of `CS_INVALID_LENGTH` is returned.

If a valid `Buffer` pointer is specified where the length specified in `Length` is sufficient, then the `Buffer` is updated with the list of CSx names available and `Length` is updated with the actual length of the string returned. If the length specified in `Length` is not sufficient to hold the contents returned in `Buffer`, then `Length` is populated with the required buffer size and a status of `CS_INVALID_LENGTH` is returned.

If a `NULL` pointer is specified for `Buffer` and a valid pointer is provided for `Length`, then the required buffer size is returned in `Length` and a status of `CS_INVALID_LENGTH` is returned. The user should allocate a buffer of the returned size and reissue the request. The user is able to also provide a large enough buffer and satisfy the request.

All input and output parameters are required for this function.

6.4.1.4 Return Value

This function returns:

a) CS_SUCCESS if there is no error and zero or more CSxes are returned; or
b) CS_LENGTH_RETURNED if NULL is specified for Buffer and a valid pointer is provided for Length and Length is returned.

6.4.1.5 Errors

The function may result in one of the following statuses:

CS_INVALID_ARG	The Length parameter provided was NULL or a non- NULL Length parameter was provided with a NULL Buffer parameter
CS_INVALID_LENGTH	The size of Buffer provided in the Length parameter was not sufficient for the list
CS_OUT_OF_RESOURCES	The function was not be completed because the library was out of resources

6.4.1.6 Notes

There may be one ore more CSxes available in the system. The caller should always check the value of Length in bytes for a non-zero value, which represents valid entries. A null terminated string is returned in Buffer if Length is non-zero. If the list contains more than one CSx entry, then the entries are comma separated. This function may still return with success when Length is zero.

The returned comma separated list of CSx names is able to be parsed and an entry is able to be selected and provided to the csOpenCSx() function to interface with the CSx.

An example source fragment implementation to return all known CSxes is:

```
length = 0;
status = csQueryCSxList(&length, NULL);
if (status != CS_INVALID_LENGTH)
    ERROR_OUT("unknown error!\n);

csx_array = malloc(length);
status = csGetCSxList(&length, &csx_array[0]);
if (status != CS_SUCCESS)
    ERROR_OUT("csGetCSxList() failed with status %d \n", status);
// process comma separated CSx list
```

6.4.2 csQueryCSFList()

This function returns zero or more CSFs available based on the query criteria.

6.4.2.1 Synopsis

CS_STATUS csQueryCSFList(const char *Path, int *Length, int *Count,
    CSFUniqueId *Buffer);

6.4.2.2 Parameters

IN Path	A string that denotes a path to a file, directory that resides on a device, a device path, or a CSx. The file/directory may indirectly refer to a namespace and partition.
IN OUT Length	A pointer to a buffer that holds or is able to hold the length of `Buffer`
OUT Count	The total count of `CSFUniqueId` data structures returned
OUT Buffer	A pointer to a buffer to hold an array of `CSFUniqueId` data structures for one or more activated CSFs if successful

6.4.2.3 Description

The `csQueryCSFList()` function fills `Buffer` with an array of `CSFUniqueId` data structures for one or more activated CSFs for the query based on `Path`. This function may return one or more `CSFUniqueId` data structures in `Buffer` that match the `Path` criteria. A `Path` set to NULL is an invalid option and a status of CS_INVALID_PATH is returned.

If a valid `Buffer` pointer is specified where the length specified in `Length` is sufficient, then the buffer is updated with the array of `CSFUniqueId` data structures available and `Length` is updated with the length of all `CSFUnqiueId` data structures returned for all functions that match `Path`. If the length specified in `Length` is not sufficient to hold the contents returned in `Buffer,` then `Length` is populated with the required buffer size and a status of `CS_INVALID_LENGTH` is returned.

If a `NULL` pointer is specified for `Buffer` and a valid pointer is provided for `Length`, then the required buffer size is returned in `Length` and a status of `CS_LENGTH_RETURNED` is returned. The user should allocate a buffer of the returned length and reissue the request. The user may also provide a large enough buffer and satisfy the request.

The Count value returned specifies the total number of CSFUniqueId data structures populated in Buffer.

All input and output parameters are required for this function.

6.4.2.4 Return Value

This function returns:

a) CS_SUCCESS if there is no error and zero or more CSxes are returned; or
b) CS_LENGTH_RETURNED if NULL is specified for Buffer and a valid pointer is provided for Length and Length is returned.

6.4.2.5 Errors

The function may result in one of the following statuses:

CS_DEVICE_NOT_PRESENT	The device name specified in the Path parameter was not present
CS_INVALID_ARG	The Length parameter provided was NULL, the Count parameter provided was NULL, or a non-null Length parameter was provided with a NULL Buffer parameter
CS_INVALID_LENGTH	The size of Buffer provided in Length parameter was not sufficient
CS_INVALID_PATH	The path to a directory or file specified in the Path parameter was not valid
CS_NO_MATCHING_DEVICE	No CSx was found matching the name specified in the Path parameter
CS_OUT_OF_RESOURCES	The function could not be completed as the library was out of resources

6.4.2.6 Notes

If the Path input specified a device path or a CSx, then the CSFUniqueId data structures returned, if any, are those available in that path. If the Path input specified a file or a directory, the query references the device path they reside on to satisfy the query.

There may be one to multiple CSFs available on any given CSx. The caller should always check the value of Length and Count for non-zero values which represents valid entries. If the Buffer contains more than one CSFUniqueId data structures, then Count specifies the total number of data structures. This function may still return with success when Length is zero.

The returned list of CSFUniqueId data structures may be parsed and a required entry may be selected for further discovery or utilized to interface with a specific CSx.

6.4.3 csGetCSxFromPath()

This function returns the CSx associated with the specified file or directory path.

6.4.3.1 Synopsis

```
CS_STATUS csGetCSxFromPath(const char *Path, unsigned int *Length,
        char *DevName);
```

6.4.3.2 Parameters

IN Path	A string that denotes a path to a file, directory that resides on a device or a device path. The file/directory may indirectly refer to a namespace and partition.
IN OUT Length	A pointer that holds or is able to hold the length of DevName
OUT DevName	Returns the qualified name to the CSx

6.4.3.3 Description

The csGetCSxFromPath() function queries the device, file, or directory path provided by Path to return the CSx associated with the specified path. If a NULL pointer is specified in the Path parameter, then all known CSxes are returned. If multiple CSxes are returned, then they are comma separated.

If a valid DevName buffer pointer is specified where the length specified in Length is sufficient, then DevName is updated with the qualified name of the CSx for access. If the length specified in Length is not sufficient to hold the contents returned in Buffer, then Length is populated with the required buffer size and a status of CS_INVALID_LENGTH is returned.

If a NULL pointer is specified for DevName and a valid pointer is provided for Length, then the requested buffer size is returned in Length and a status of

CS_LENGTH_RETURNED is returned. The user should allocate a buffer of the returned length and reissue the request. The user may also provide a large enough buffer and satisfy the request.

All input and output parameters are required for this function.

6.4.3.4 Return Value

This function returns:

a) CS_SUCCESS if there is no error and zero or more CSxes are returned; or

b) CS_LENGTH_RETURNED if NULL is specified for Buffer and a valid pointer is provided for Length and Length is returned.

6.4.3.5 Errors

The function may result in one of the following statuses:

CS_INVALID_ARG	The Length parameter provided was NULL or a non-null Length parameter was provided with a NULL Buffer parameter
CS_INVALID_LENGTH	The size of Buffer provided in Length parameter was not sufficient
CS_INVALID_PATH	The path to a directory or file specified in the Path parameter was not valid
CS_NO_MATCHING_DEVICE	No CSx was found associated with the device specified in the Path parameter
CS_OUT_OF_RESOURCES	The function could not be completed as the library was out of resources

6.4.3.6 Notes

The Path parameter denotes the path to a device, filename or directory on a Linux filesystem. If the path specified is partial, then the path is resolved to its full path internally before mapping the device pair. This function works with most typical Linux file systems (e.g., ext3, ext4, and xfs) that are mounted on an underlying device without any RAID indirections. This function returns CS_NO_MATCHING_DEVICE for such inputs.

The returned DevName is qualified to be used with the the csOpenCSx() function to interface with the CSE.

An example source fragment implementation would be:

```
status = csGetCSxFromPath(my_file_path, &length, &csx_array[0]);
if (status != CS_SUCCESS) {
```

```
    ERROR_OUT("The specified path %s returned an error %d\n", my_file_path, status);}
// open device, initialize CSF, and pre-allocate buffers
status = csOpenCSx(csx_array[0], &dev_context, &dev);
…
```

6.5 Access

These set of functions are used to access a CSE. The user is able to utilize the discovery functions to find the CSE through the Storage/filesystem pair.

6.5.1 csOpenCSx()

Return a handle to the CSx associated with the specified device name.

6.5.1.1 Synopsis

```
CS_STATUS csOpenCSx(const char *DevName, void *DevContext,
        CS_DEV_HANDLE *DevHandle);
```

6.5.1.2 Parameters

IN DevName	A string that denotes the full name of the device
IN DevContext	A user specified context to associate with the device for future notifications
OUT DevHandle	Returns the handle to the CSE device

6.5.1.3 Description

The `csOpenCSx()` function opens the CSx and provides a handle for future usages to the user.

If a valid `DevName` is specified and available, a handle to the CSx is returned if all other parameters are valid.

All input and output parameters are required for this function.

6.5.1.4 Return Value

This function returns `CS_SUCCESS` if there is no error and the specified CSx was found.

6.5.1.5 Errors

The function may result in one of the following statuses:

CS_DEVICE_NOT_PRESENT	The device name specified in `DevName` parameter was not present
CS_INVALID_ARG	The `DevHandle` parameter provided was NULL

6.5.2 csCloseCSx()

Close a CSx previously opened and associated with the specified handle.

6.5.2.1 Synopsis

CS_STATUS csCloseCSx(CS_DEV_HANDLE DevHandle);

6.5.2.2 Parameters

IN DevHandle	Handle to CSx

6.5.2.3 Description

A valid `DevHandle` is required to be provided for this function. If the CSx is open, then the CSx is closed and all outstanding requests are terminated.

All input and output parameters are required for this function.

6.5.2.4 Return Value

This function returns `CS_SUCCESS` if there is no error and the CSx was found as specified.

6.5.2.5 Errors

The function may result in one of the following statuses:

CS_INVALID_HANDLE	The `DevHandle` parameter was not a valid handle

6.5.3 csRegisterNotify()

Register a callback function to be notified based on various computational storage events across all CSxes.

This is an optional function.

6.5.3.1 Synopsis

CS_STATUS csRegisterNotify(const char *DevName, u32 NotifyOptions,
csDevNotificationFn NotifyFn);

6.5.3.2 Parameters

IN DevName	A string that denotes a specific CSE or CSx to provide notifications for. If NULL, then all CSEs and CSxes are registered
IN NotifyOptions	Denotes the notification types to register (see 6.3.8)
IN NotifyFn	A user specified callback notification function

6.5.3.3 Description

The `csRegisterNotify()` function registers the provided callback for notifications based on options selected in `NotifyOptions` by the user.

If `DevName` specifies a valid CSE or CSx, then the notifications types are added to the registered notification types for the specified CSE or CSx. If `DevName` is `NULL`, then the notifications are registered across all CSxes and CSEs.

All input parameters are required for this function.

6.5.3.4 Return Value

This function returns `CS_SUCCESS` if there are no errors.

6.5.3.5 Errors

The function may result in one of the following statuses:

CS_DEVICE_NOT_PRESENT	The device name specified in `DevName` parameter was not present
CS_INVALID_ARG	The device name specified in `DevName` parameter was not valid, the `NotifyOptions` parameter was zero, or the `NotifyFn` parameter was NULL
CS_INVALID_OPTION	One or more of the selected options specified in the `NotifyOptions` parameter were not supported
CS_OUT_OF_RESOURCES	The function could not be completed as the library was out of resources

6.5.3.6 Notes

The callback is invoked by the function subsystem to provide notifications asynchronously based on notification options provided at registration time. Callbacks may be invoked for different types of notifications and errors, some of which may be fatal (i.e., the device is not able to recover from its error state). The caller acts upon these notifications with appropriate actions.

6.5.4 csDeregisterNotify()

Deregister a previously registered callback function for notifications for computational storage events. A callback function may have been previously registered using the `csRegisterNotify()` function.

This is an optional function.

6.5.4.1 Synopsis

```
CS_STATUS csDeregisterNotify(const char *DevName, u32 NotifyOptions,
        csDevNotificationFn NotifyFn);
```

6.5.4.2 Parameters

IN DevName	A string that denotes a specific CSE or CSx to deregister notifications from. If NULL, then all CSEs and CSxes are deregistered
IN NotifyOptions	Denotes the notifications to deregister from (see 6.3.8)
IN NotifyFn	The callback notification function previously registered

6.5.4.3 Description

The `csDeregisterNotify()` function removes a previously provided callback for notifications from one or more CSEs or CSxes.

If a `DevName` specifies a valid CSE or CSx, the notifications are deregistered for the specified CSE or CSx. If `DevName` is NULL, then the notifications specified by `NotifyOptions` are deregistered for all CSxes and CSEs.

All input parameters are required for this function.

6.5.4.4 Return Value

This function returns CS_SUCCESS if there are no errors.

6.5.4.5 Errors

The function may result in one of the following statuses:

CS_DEVICE_NOT_PRESENT	The device name specified in DevName parameter was not present
CS_INVALID_ARG	The device name specified in DevName parameter was not valid, zero NotifyOptions parameter were provided, or the NotifyFn parameter was NULL
CS_OUT_OF_RESOURCES	The function could not be completed as the library was out of resources

6.6 AFDM management

6.6.1 csAllocMem()

Allocates memory from the FDM for the requested size in bytes.

6.6.1.1 Synopsis

```
CS_STATUS csAllocMem(CS_DEV_HANDLE DevHandle, int Bytes,
        const CsMemFlags *MemFlags, CS_MEM_HANDLE *MemHandle,
        CS_MEM_PTR *VAddressPtr);
```

6.6.1.2 Parameters

IN DevHandle	Handle to CSx
IN Bytes	Length in bytes of FDM to allocate
IN MemFlags	Options for allocating FDM (see 6.3.5.3.3.1)
OUT MemHandle	Pointer to a buffer to hold the memory handle once allocated
OUT VAddressPtr	Pointer to a buffer to hold the virtual address of device memory allocated in host system address space. This is

optional and may be NULL if memory is not required to be mapped

6.6.1.3 Description

The csAllocMem() function allocates requested memory from FDM.

If a valid MemHandle pointer is specified, MemHandle is updated with the handle to the AFDM. If a valid VAddressPtr pointer is specified, the AFDM is mapped into the user's virtual address space in host memory. Only CSxes with FDM host visible capability may use the VAddressPtr parameter. See section 6.3.5.3.1.9 for the capability details.

The values set in the MemFlags data structure describe how FDM is allocated. The csAllocMem() function allocates memory specified by the FDMId in the MemFlags parameter and only uses CS_FDM_CLEAR and CS_FDM_FILL, as defined in section 6.3.5.2.7. If CS_FDM_CLEAR is selected, then the contents of AFDM is cleared. If CS_FDM_FILL is specified, then the contents of AFDM are populated with the value in FillValue. The FillValue field is only valid if CS_FDM_CLEAR is specified and ignored otherwise. A value of zero in the Flags field specifies that no option was selected for MemFlags and AFDM is not modified on allocation.

All input parameters are required for this function.

6.6.1.4 Return Value

This function returns CS_SUCCESS if there were no errors and device memory was successfully allocated.

6.6.1.5 Errors

The function may result in one of the following statuses:

CS_DEVICE_NOT_AVAILABLE	The device referenced by the DevHandle parameter was not available for processing
CS_COULD_NOT_MAP_MEMORY	The requested memory allocation could not be mapped
CS_INVALID_ARG	The DevHandle parameter was NULL, the Bytes parameter was zero, the MemFlags parameter was NULL, or the MemHandle parameter was NULL

CS_INVALID_FDM_ID	The specified FDMId in the `MemFlags` parameter was not valid
CS_INVALID_HANDLE	The `DevHandle` parameter was invalid
CS_INVALID_OPTION	One or more options specified in `MemFlags` were not valid
CS_NOT_ENOUGH_MEMORY	There was insufficient memory in the FDM to satisfy the requested allocation
CS_OUT_OF_RESOURCES	The function could not be completed as the library was out of resources

6.6.1.6 Notes

AFDM is allocated using this function. AFDM is allocated on a host page size granularity and is rounded off for other values that are not multiples of this size. If a virtual address pointer is requested, the function shall return a virtual address pointer that is host page aligned.

The optional parameter `VAddressPtr` should only be used when the host application requires data transfer between storage and AFDM. For all other cases this field should be set to NULL and the `MemHandle` returned from this function call should be used instead. These details are summarized as follows:

a) If the host application wants to use the direct p2p capability between storage and AFDM, then the host provides VAddressPtr as the buffer to filesystem read/write requests. Care should be taken that no buffering is enabled executing through a filesystem path by specifying the O_DIRECT flag when a file is opened. For those filesystems that do not provide such an interface, an appropriate mechanism should be used to keep data coherent. See section 6.7 for additional details; and
b) For usages where the host applications require data transfer between host memory and device memory, this parameter is not required and should be set to NULL. See usage of csQueueCopyMemRequest() in section 6.8.1.

6.6.2 csFreeMem()

Frees AFDM for the memory handle specified.

6.6.2.1 Synopsis

```
CS_STATUS csFreeMem(CS_MEM_HANDLE MemHandle,
    const CsMemFlags *MemFlags);
```

6.6.2.2 Parameters

IN MemHandle	Handle to AFDM
IN MemFlags	Options to specify while freeing FDM (see 6.3.5.3.3.1)

6.6.2.3 Description

The csFreeMem() function frees previously requested AFDM.

If a valid MemHandle value is specified, the memory represented by MemHandle is freed and returned to the FDM. Any memory mappings created by the allocate call are also released and freed.

The values set in MemFlags describe how to handle AFDM when AFDM is freed. The csFreeMem() function only uses CS_FDM_CLEAR and CS_FDM_FILL as defined in section 6.3.5.2.7. If CS_FDM_CLEAR is selected, then the contents of AFDM represented by MemHandle is cleared. If CS_FDM_FILL is specified, then the contents of AFDM represented by MemHandle is populated with the value in FillValue. The FillValue field is only valid if CS_FDM_FILL is specified and ignored otherwise. A value of zero in the Flags field specifies that no option is selected for MemFlags and AFDM is not modified when AFDM is freed.

The FDMId field in MemFlags does not apply and is ignored for this function.

All input parameters are required for this function.

6.6.2.4 Return Value

This function returns CS_SUCCESS if there is no error.

6.6.2.5 Errors

The function may result in one of the following statuses:

CS_COULD_NOT_UNMAP_MEMORY	The memory specified by the MemHandle parameter was not able to be unmapped and freed
CS_DEVICE_NOT_AVAILABLE	The value in the MemHandle parameter was valid and the device that the referenced memory was on, was not available for processing the request
CS_INVALID_ARG	The MemHandle parameter was NULL or the MemFlags parameter was NULL

CS_INVALID_HANDLE	The `MemHandle` parameter was not valid (e.g., the `MemHandle` did not exist)
CS_MEMORY_IN_USE	The memory specified by the `MemHandle` parameter was in use and cannot be freed
CS_OUT_OF_RESOURCES	The function could not be completed as the library was out of resources

6.6.2.6 Notes

The caller should ensure that no outstanding transactions are present on the memory handle being freed. If there outstanding transactions, then the request returns CS_MEMORY_IN_USE.

6.6.3 csInitMem()

Initialize AFDM contents for the memory handle specified.

6.6.3.1 Synopsis

CS_STATUS csInitMem(CS_MEM_HANDLE MemHandle, unsigned long ByteOffset, int Bytes, const CsMemFlags *MemFlags);

6.6.3.2 Parameters

IN MemHandle	Handle to AFDM
IN ByteOffset	Offset at which to start initialization
IN Bytes	Number of bytes to initialize
IN MemFlags	Options to initialize AFDM (see 6.3.5.3.3.1)

6.6.3.3 Description

The `csInitMem()` function initializes the contents of a previously requested AFDM with the specified details.

If a valid `MemHandle` value is specified, the contents of the memory represented by the handle is initialized. The option specified in the `MemFlags` parameter is applied to the specified AFDM for number of bytes specified by `Bytes` starting at `ByteOffset`.

The ByteOffset and Bytes parameters is required to specify valid values for the AFDM being initialized. This function returns an error if the offset plus the number of bytes specified is greater than the size of the AFDM.

The MemFlags parameter for the csInitMem() function only supports CS_FDM_CLEAR and CS_FDM_FILL for the Flags field as defined in section 6.3.5.2.7. If CS_FDM_CLEAR is selected, then the AFDM referenced by MemHandle is cleared. If CS_FDM_FILL is specified, then the value in the FillValue field is used to populate the AFDM referenced by MemHandle. The FillValue field is only valid if CS_FDM_FILL is specified and ignored otherwise. A value of zero in the Flags field specifies that the contents of AFDM represented by MemHandle not be initialized. The FDMId field in MemFlags is ignored for this function.

All input parameters are required for this function.

6.6.3.4 Return Value

This function returns CS_SUCCESS if there is no error.

6.6.3.5 Errors

The function may result in one of the following statuses:

CS_DEVICE_NOT_AVAILABLE	The device, on which the memory referenced by the MemHandle parameter exists, was not available for processing
CS_INVALID_ARG	The MemHandle parameter was NULL or the MemFlags was NULL
CS_INVALID_HANDLE	The MemHandle parameter was not valid (e.g., the MemHandle does not exist)
CS_INVALID_LENGTH	The ByteOffset parameter exceeded the length of AFDM, the Bytes parameter was zero, or the sum of the Bytes parameter and the ByteOffset parameters was greater than the size of AFDM
CS_INVALID_OPTION	The FillValue field in the MemFlags parameter was not valid
CS_OUT_OF_RESOURCES	The function could not be completed as the library was out of resources

6.6.3.6 Notes

The caller should ensure that the memory initialization requested at AFDM's offset and bytes has no outstanding transactions in progress. Doing so may incur unknown results.

6.7 Storage IOs

I/O requests to and from storage devices are typically orchestrated through existing filesystems and block subsystem interfaces. P2P transfers between storage and CSxes may be achieved through filesystem read/write calls or using the `csQueueStorageRequest()` function. For filesystem usage, the AFDM is allocated with virtual address mapping, and this address pointer is then passed along to the filesystem/block subsystem. This allows the data to be loaded directly into AFDM from storage and vice versa. Only CSxes with FDM host visible capability may use the filesystem access path.

For more advanced usages, P2P access alone may not be able to satisfy a user request. The following are examples where P2P with a filesystem may not work:

- the CSx does not support host visible FDM; and
- the user requires remote CSx access.

6.7.1 csQueueStorageRequest()

Queues a storage I/O request to the device.

6.7.1.1 Synopsis

CS_STATUS csQueueStorageRequest(const CsStorageRequest *Req, void *Context,
csQueueCallbackFn CallbackFn, CS_EVT_HANDLE EventHandle,
CS_REQ_HANDLE *ReqHandle, u64 *CompValue);

6.7.1.2 Parameters

IN Req	Structure to the storage request
IN Context	A user specified context for the storage request when asynchronous. The parameter is required only if `CallbackFn` or `EventHandle` is specified.
IN CallbackFn	A callback function if the request is required to be asynchronous.
IN EventHandle	A handle to an event previously created using the `csCreateEvent()` function. This value may be NULL if

	the `CallbackFn` parameter is specified to be a valid value or if the request is synchronous.
OUT ReqHandle	A pointer to a buffer to hold the request handle if successful. The received handle is able to be used to abort this request using the `csAbortRequest()` function. This is an optional parameter and depends on the implementation.
OUT CompValue	Additional completion value provided as part of completion. This may be optional depending on the implementation.

6.7.1.3 Description

The `csQueueStorageRequest()` function queues a storage request to the device.

A valid `Req` structure (see6.3.5.3.4.1) is required to initiate the storage I/O operation. All fields in the `Req` structure are required and describe the source and destination details. The request may be performed synchronously or asynchronously. To be performed synchronously, the parameters `CallbackFn` and `EventHandle` should be set to NULL and `Context` is ignored. To be performed asynchronously, either a callback is required to be specified in `CallbackFn` or an event handle is required to be specified in `EventHandle`. It is an error to specify both of these parameters. If `Context` is specified, it is returned in the asynchronous completion path.

An optional pointer may be specified to receive a `ReqHandle` for the request. The `ReqHandle` allows the request to subsequently be aborted. For asynchronous operation, if a valid pointer is specified, the pointer is updated with a handle to the submitted request. A synchronous operation ignores this parameter.

For `EventHandle`, see the `csCreateEvent()` function for usage.

6.7.1.4 Return Value

If there are no errors, then for:

a) a synchronous data transfer operation a status of CS_SUCCESS is returned; and
b) an asynchronous data transfer operation a status of CS_QUEUED is returned.

6.7.1.5 Errors

The function may result in one of the following statuses:

CS_DEVICE_NOT_AVAILABLE	The device referenced by the `DevHandle` field in the `Req` parameter was not available for processing the request
CS_INVALID_ARG	The `Req` parameter was NULL or the `CallbackFn` and `EventHandle` were both set
CS_INVALID_HANDLE	The `EventHandle` parameter or the `DevHandle` field in the `Req` parameter was not valid
CS_INVALID_OPTION	One or more options selected in the `Req` data structure parameter were invalid
CS_UNKNOWN_MEMORY	The memory referenced by the memory handle in the `DevMem` data structure of the `Req` parameter was not valid

This function may provide additional completion value returned in `CompValue`.

6.8 CSx data movement

The application is able to copy data from host memory to AFDM or from AFDM to host memory using this function call.

6.8.1 csQueueCopyMemRequest()

Copies data between host memory and AFDM in the direction requested.

6.8.1.1 Synopsis

```
CS_STATUS csQueueCopyMemRequest(const CsCopyMemRequest *CopyReq,
        void *Context, csQueueCallbackFn CallbackFn,
        CS_EVT_HANDLE EventHandle, CS_REQ_HANDLE *ReqHandle,
        u64 *CompValue);
```

6.8.1.2 Parameters

IN CopyReq	A request structure that describes the source and destination details of the copy request

IN Context	A user specified context for the copy request when asynchronous. The parameter is required only if `CallbackFn` or `EventHandle` is specified.
IN CallbackFn	A callback function if the copy request is required to be asynchronous.
IN EventHandle	A handle to an event previously created using the `csCreateEvent()` function. This value may be NULL if the `CallbackFn` parameter is specified to be valid value or if also set to NULL when the request is required to be synchronous.
OUT ReqHandle	A pointer to a buffer to hold the request handle if successful. The received handle is able to be used to abort this request using the `csAbortRequest()` function.This is an optional parameter and depends on the implementation.
OUT CompValue	Additional completion value provided as part of completion. This may be optional depending on the implementation.

6.8.1.3 Description

The `csQueueCopyMemRequest()` function copies data between device memory and host memory in the specified direction.

A valid `CopyReq` structure (see 6.3.5.3.3.3) is required to initiate the copy operation. All fields in the `CopyReq` structure are required and describe the source and destination details. To perform the request synchronously, the parameters `CallbackFn` and `EventHandle` should be set to NULL and `Context` is ignored. To perform the request asynchronously, either a callback is required to be specified in `CallbackFn` or an event handle is required to be specified in `EventHandle`. It is an error to specify both of these parameters. If `Context` is specified, `Context` is returned in the asynchronous completion path. See notes for details.

An optional pointer may be specified to receive a `ReqHandle` for the request. The `ReqHandle` allows the request to subsequently be aborted. For asynchronous operation, if a valid pointer is specified, the pointer is updated with a handle to the submitted request. A synchronous operation ignores this parameter.

6.8.1.4 Return Value

If there are no errors, then for:

a) a synchronous data transfer operation a status of CS_SUCCESS is returned; and
b) an asynchronous data transfer operation a status of CS_QUEUED is returned.

6.8.1.5 Errors

The function may result in one of the following statuses:

CS_DEVICE_NOT_AVAILABLE	One or more devices referenced by the `MemHandle` field(s) in the `CopyReq` parameter were not available for processing the request
CS_INVALID_ARG	The `CopyReq` parameter was NULL; or the `CallbackFn` and `EventHandle` were both set
CS_INVALID_HANDLE	The `EventHandle` parameter was not valid or one or more of the `MemHandle` fields in the `CopyReq` parameter were not valid
CS_INVALID_OPTION	The `HostVAddress field` and `SrcDevMem` field in the `CopyReq` parameter were both set to non-zero values
CS_UNKNOWN_MEMORY	The memory referenced by `MemHandle` was unknown

This function may provide additional completion value returned in `CompValue`.

6.8.1.6 Notes

The `CsCopyMemRequest` structure describes the copy request with the host memory details, device memory details, and the size in the `Bytes` field that are required to be copied. The `Type` field describes the direction for the memory copy.

The `ByteOffset` field in `CsDevAFDM` specifies an offset in bytes from the start of the AFDM represented by the `MemHandle` field. Copy operations may be conducted between a host memory address and AFDM or between two AFDMs. Host memory is always represented by the `HostVAddress` field irrespective of the direction. This field should be set to NULL if not specified.

The copy operation may be requested to be synchronous or asynchronous. If synchronous, then all other inputs other than `CopyReq` should be set to NULL. If asynchronous, then either the `CallbackFn` or the `EventHandle` is required to be set to a valid value. It is an error for both the `CallbackFn` and the `EventHandle` to be set.

An example source fragment implementation to copy from host memory to device memory is:

```
// copy 4kb from host buffer to offset 0 of device memory handle synchronously
copyReq.Type = CS_COPY_TO_DEVICE;
copyReq.u.HostVAddress = &buffer;
copyReq.DevMem.MemHandle = devMem[0];
copyReq.DevMem.ByteOffset = 0;
copyReq.Bytes = 4096;
// block till copy is complete
status = csQueueCopyMemRequest(&copyReq, NULL, NULL, NULL, Null, NULL);
if (status != CS_SUCCESS)
    ERROR_OUT("csQueueCopyMemRequest() failed with status %d!\n", status);
…
```

6.9 CSF scheduling

CSxes provide one or more CSFs to which compute work may be scheduled. These functions require a mechanism to invoke them and collect their results.

The following functions provide the functionality to invoke one or more CSFs.

6.9.1 csGetCSFId()

Fetches the CSF details specified by name for scheduling compute offload tasks.

6.9.1.1 Synopsis

CS_STATUS csGetCSFId(CS_DEV_HANDLE DevHandle, const char *CSFName,
    u64 GlobalId, int *Length, int *Count, CSFIdinfo *Buffer);

6.9.1.2 Parameters

IN DevHandle	Handle to CSx
IN CSFName	A pre-specified function name, if GlobalId is not specified
IN GlobalId	Global Identifier, if CSFName is not specified
IN OUT Length	A pointer to a buffer that holds or is able to hold the length of `Buffer`
OUT Count	Count of CSFIdInfo structures returned
OUT Buffer	A pointer to a buffer to hold an array of `CSFIdInfo` data structures for one or more CSFs if successful that contains `FDMId`, performance, and power details

6.9.1.3 Description

The `csGetCSFId()` function returns one or more `CSFIdInfo` data structures in `Buffer` when the length specified in `Length` is sufficient to satisfy the request. The `CSFName` or `GlobalId` should be a valid value that is available in the CSx specified the `DevHandle` parameter.

This function returns an error if:

a) the specified CSFName or GlobalId is not found; or
b) both CSFName and GlobalId are specified.

CSFs may be queried by either `CSFName` or `GlobalId`. If `CSFName` is specified by a valid NULL terminated string, then `GlobalId` should be set to zero. If `GlobalId` is specified, then `CSFName` should be set to NULL.

If a valid `Buffer` pointer is specified where the length specified in `Length` is sufficient, then `Buffer` is updated with an array of available `CSFIdInfo` data structures and `Length` is updated with the actual length of data returned in `Buffer`. If the length specified in `Length` is not sufficient to hold the contents returned in `Buffer`, then `Length` is populated with the required length and an error status is returned.

If a `NULL` pointer is specified for `Buffer` and a valid pointer is provided for `Length`, then the required buffer length is returned in `Length`. The user should allocate a buffer of the returned length and reissue the request.

The `Count` value returned specifies the total number of `CSFIdInfo` data structures populated in `Buffer`.

All input and output parameters are required for this function.

6.9.1.4 Return Value

This function returns:

a) `CS_SUCCESS` if there is no error and zero or more `CSFIdInfo` data structures are returned; or
b) `CS_LENGTH_RETURNED` if `NULL` is specified for Buffer and a valid pointer is provided for `Length` and Length is returned.

6.9.1.5 Errors

The function may result in one of the following statuses:

CS_DEVICE_NOT_AVAILABLE	The CSx specified by the `DevHandle` parameter was valid but not available
CS_INVALID_ARG	The `DevHandle` parameter was NULL, both `CSFName` and `GlobalId` parameters were specified, both `CSFName` and `GlobalId` parameters were NULL, `Length` parameter was NULL, or `Count` parameter was NULL

CS_INVALID_CSF_NAME	The specified `CSFName` was not found
CS_INVALID_GLOBAL_ID	The specified `GlobalId` was not found
CS_INVALID_HANDLE	The `DevHandle` parameter was not a valid handle
CS_INVALID_LENGTH	The size of `Buffer` provided in the `Length` parameter was not sufficient

6.9.1.6 Notes

Any compute work that is required to be run on a CSx first requires the associated CSFs to be configured. A list of configured CSFs may be queried through the `csQueryCSFList()` function.

This function should be called prior to any compute work being scheduled. The data returned in `Buffer` may contain an array of `CSFIdInfo` data structures. The `CSFId` data field returned uniquely identifies the CSF and is used for scheduling work. The `RelativePerformance` data field and `RelativePower` data field help differentiate between multiple CSF instances, if returned by this function. The `Count` data field denotes the number of instances for this CSF and determines the parallelism available.

The `NumFDMs` data field provides the details of the FDMs that are accessible by the CSF in `CSFIdInfo` data structure and `FDMList` is a pointer to the list of FDMs. Each FDM entry contains the `FDMId` that may be used to allocate FDM using the `csAllocMem()` function while the `RelativePerformance` and `RelativePower` data fields help differentiate between FDMs.

6.9.2 csAbortRequest()

Aborts the queued request that is specified by the request handle.

6.9.2.1 Synopsis

CS_STATUS csAbortRequest(CS_REQ_HANDLE ReqHandle);

6.9.2.2 Parameters

IN ReqHandle	Handle to the outstanding request to abort.

6.9.2.3 Description

The `csAbortRequest()` function aborts the specified request. The `ReqHandle` parameter represents a request submitted using one of the `csQueueStorageRequest()` function, the `csQueueCopyMemRequest()` function, the `csQueueComputeRequest()` function, or the `csQueueBatchRequest()` function. If successful, the outstanding request is canceled from the queue and completed in error.

6.9.2.4 Return Value

A status value of `CS_SUCCESS` is returned if no errors were encountered in aborting the CSF.

6.9.2.5 Errors

The function may result in one of the following statuses:

CS_DEVICE_NOT_AVAILABLE	The CSx refernced by the `ReqHandle` parameter was not responding
CS_FATAL_ERROR	A fatal error has occurred and the request was not able to be aborted
CS_INVALID_HANDLE	The `ReqHandle` parameter was not valid
CS_UNSUPPORTED	This function was not supported

6.9.2.6 Notes

Use this function to abort a queued task that may no longer be valid or is misbehaving.

6.9.3 csQueueComputeRequest()

Queues a compute offload request to the device to be executed synchronously or asynchronously in the device.

6.9.3.1 Synopsis

CS_STATUS csQueueComputeRequest(const CsComputeRequest *Req,
    void *Context, csQueueCallbackFn CallbackFn,
    CS_EVT_HANDLE EventHandle, CS_REQ_HANDLE *ReqHandle,
    u64 *CompValue);

6.9.3.2 Parameters

IN Req	A request structure that describes the CSE function and its arguments to queue.

IN Context	A user specified context for the queue request when asynchronous. The parameter is required only if `CallbackFn` or `EventHandle` is specified.
IN CallbackFn	A callback function if the queue request is required to be asynchronous.
IN EventHandle	A handle to an event previously created using `csCreateEvent`. This value may be NULL if `CallbackFn` parameter is specified to be valid value or if also set to NULL when the request is required to be synchronous.
OUT ReqHandle	A pointer to a buffer to hold the request handle if successful. The received handle is able to be used to abort this request using the `csAbortRequest()` function.This is an optional parameter and depends on the implementation.
OUT CompValue	Additional completion value provided as part of completion. This may be optional depending on the implementation.

6.9.3.3 Description

The `csQueueComputeRequest()` function queues a CSF request to the CSx. The inputs and outputs for the CSF are specified in the `Req` data structure. The request may be performed synchronously or asynchronously. To perform the request synchronously, the parameters `CallbackFn` and `EventHandle` should be set to NULL and `Context` is ignored. To perform the request asynchronously, either a callback is required to be specified in `CallbackFn` or an event handle is required to be specified in `EventHandle`. It is an error to specify both of these parameters. If `Context` is specified, `Context` is returned in the asynchronous completion path.

An optional pointer may be specified to receive a `ReqHandle` for the request to allow the request to be aborted. For asynchronous operation, if a valid pointer is specified, it is updated with a handle for the submitted request. A synchronous operation ignores this parameter.

For more information on callback usage, see 6.3.8.

For more information on using events for polling see 6.11.3.

6.9.3.4 Return Value

if there are no errors, then for:

a) a synchronous queue operation CS_SUCCESS is returned; and
b) an asynchronous queue operation CS_QUEUED is returned.

6.9.3.5 Errors

The function may result in one of the following statuses:

CS_DEVICE_NOT_AVAILABLE	A CSx referenced by the CSFId or MemHandle field in the Req parameter was not available for processing
CS_ERROR_IN_EXECUTION	There was an error in execution of the specified CSFId
CS_INVALID_ARG	The Req parameter was NULL or the CallbackFn and EventHandle were both set
CS_INVALID_CSF_ID	The specified CSFId in the Req parameter was not a valid Id
CS_INVALID_HANDLE	The EventHandle parameter was not valid or the MemHandle field in CsComputeArg was not valid
CS_INVALID_OPTION	The HostVAddress and SrcDevMem fields were both set
CS_UNKNOWN_MEMORY	The memory referenced by the MemHandle field in CsComputeArg was unknown

This function may provide additional completion value returned in CompValue.

6.9.3.6 Notes

The CSF is required to be loaded first and its handle populated in the Req structure.

This is a generic queueing function for any type of CSF. It is the responsibility of the caller to ensure that the number of arguments and their individual values map correctly to the CSF.

The data structure CsComputeRequest (see 6.3.5.3.5.1) describes the function the request should be issued to and its input arguments. The field NumArgs defines the number of arguments that are required to be issued to the function. The user should ensure that these match actual function inputs.

See the csQueueCopyMemRequest() function (see 6.8.1) for DevMem field details and requirements on the CallbackFn and EventHandle inputs. An EventHandle is utilized only by user space applications. Kernel space applications such as drivers and filesystems use the CallbackFn.

For EventHandle, see the csCreateEvent() function for usage.

6.9.4 csCSEEDownload()

Downloads a specified CSEE resource. It is implementation specific as to how the downloaded resource is secured.

This is a privileged function.

6.9.4.1 Synopsis

```
CS_STATUS csCSEEDownload(CS_DEV_HANDLE DevHandle,
        const CsCSEEDownloadInfo *Info, u32 *CSEEId);
```

6.9.4.2 Parameters

IN DevHandle	Handle to CSx
IN Info	A pointer to a buffer that holds the CSEE resource details to download
OUT CSEEId	A pointer to a buffer to hold the identifier to the downloaded CSEE resource

6.9.4.3 Description

The csCSEEDownload() function downloads a CSEE using the details in Info. The Info parameter provides the details of download contents (e.g., the CS_CSEE_RESOURCE_TYPE and CS_RESOURCE_SUBTYPE. Additional details on these fields are provided in section 6.3.5.2.2 and 6.3.5.2.4. On a successful download, a CSEEId for the downloaded CSEE is returned. This value may be used to configure the downloaded resource using the csConfig() function.

All parameters are required for this function.

6.9.4.4 Return Value

CS_SUCCESS is returned if there are no errors.

6.9.4.5 Errors

The function may result in one of the following statuses:

CS_DEVICE_NOT_AVAILABLE	The device specified by the DevHandle parameter was not available
CS_INVALID_ARG	The DevHandle parameter was NULL, Info was NULL, or CSEEId was NULL
CS_INVALID_HANDLE	The DevHandle parameter was not valid
CS_INVALID_OPTION	An option selected in the Info parameter was not valid
CS_LOAD_ERROR	There was an error in downloading
CS_NO_PERMISSIONS	There were insufficient user permissions to satisfy this request
CS_NOT_ENOUGH_MEMORY	There was insufficient memory to satisfy this request
CS_UNSUPPORTED	This function is not supported
CS_UNSUPPORTED_INDEX	The specified index in Info was not supported
CS_UNSUPPORTED_TYPE	The specified type in Info not supported

6.9.4.6 Notes

CSxes that contain a CSE that is not capable of accepting a downloaded CSEE fail this function (e.g., CSx devices that only have fixed functionality).

6.9.5 csHelperSetComputeArg()

Helper function that is able to optionally be used to set an argument for a compute request.

6.9.5.1 Synopsis

```
void csHelperSetComputeArg(CsComputeArg *ArgPtr,
        CS_COMPUTE_ARG_TYPE Type, ...);
```

Parameters

IN ArgPtr	A pointer to the argument in `CsComputeRequest` to be set.
IN Type	The argument type to set. This value may be one of the enumerated type values.
IN <…>	One or more variables that make up the argument by type.

6.9.5.2 Description

The `csHelperSetComputeArg()` function is a helper function that sets an argument for a compute request. A compute request may have one or more arguments. Each argument may have one or more inputs that describe it. This function sets up the argument with minimal code.

This function does not validate inputs.

6.9.5.3 Return Value

No status is returned from this function since it does not change any values.

6.9.5.4 Notes

The helper function may optionally be used to setup individual arguments to a compute request as shown in the following example code snippet. It helps replace the commented code when applied.

```
// setup compute request with 3 arguments
req->CSFId = functId;
req->NumArgs = 3;
argPtr = &req->Args[0];
csHelperSetComputeArg(&argPtr[0], CS_AFDM_TYPE, inMemHandle, inMemOffset);
csHelperSetComputeArg(&argPtr[1], CS_32BIT_VALUE_TYPE, 16384);
csHelperSetComputeArg(&argPtr[2], CS_AFDM_TYPE, outMemHandle, 0);
/* code it replaces
argPtr[0].Type = CS_AFDM_TYPE;                  // input data buffer
argPtr[0].u.DevMem.MemHandle = inMemHandle;
argPtr[0].u.DevMem.ByteOffset = inMemOffset;
argPtr[1].Type = CS_32BIT_VALUE_TYPE;           // size
argPtr[1].u.Value32 = 16384;
argPtr[2].Type = CS_AFDM_TYPE;                  // output data buffer
argPtr[2].u.DevMem.MemHandle = outMemHandle;
argPtr[2].u.DevMem.ByteOffset = 0;
*/
```

6.10 Batch scheduling

For offload work that involves more than one step with functions, batch scheduling aids in queuing such requests. Batching may involve serializing multiple requests pipelined to execute one after another or parallelizing them to execute together, provided the required hardware resources are available.

The batch functions are optional and may return CS_UNSUPPORTED.

The process of scheduling batched requests helps in the following ways:

a) Minimize host orchestration sub-tasks and associated latency costs;
b) Minimize host CPU context switches;
c) Simplify the number of steps involved in processing user data; and
d) Reduce overall latency of the intended compute work.

Batch request processing may be conducted with the `csAllocBatchRequest()` function (see 6.10.1), the `csFreeBatchRequest()` function (see 6.10.2), the `csConfigureBatchEntry()` function (see 6.10.3), the `csHelperResizeBatchRequest()` function (see 6.10.4), and the `csQueueBatchRequest()` function (see 6.10.5). A batch operation is setup by first creating a batch request and then populating the batch request with the list of requests. Once setup, the operation is able to be queued using the `csQueueBatchRequest()` function. Batch operations are identified by the batch handle and are able to be reused once a queued request is complete. Optionally, entries added to the batch request are able to be reconfigured as required for successive IOs.

6.10.1 csAllocBatchRequest()

Allocates a batch handle that may be used to submit batch requests. The handle resource may be set up with the individual requests that are to be batch processed. The allocation may be requested for serial, parallel, or hybrid batched request flows that support storage, compute, and data copy requests all in one function.

6.10.1.1 Synopsis

CS_STATUS csAllocBatchRequest(CS_BATCH_MODE Mode, int MaxReqs,
    CS_BATCH_HANDLE *BatchHandle);

6.10.1.2 Parameters

IN Mode	The requested batch mode namely, serial, parallel or hybrid.
IN MaxReqs	The maximum number of requests the caller perceives added to this batch resource. This parameter provides a hint to the sub-system for resource management.
OUT BatchHandle	The created handle for batch request processing if successful.

6.10.1.3 Description

The `csAllocBatchRequest()` function creates a batch request handle resource that may be used to queue more than one request later.

6.10.1.4 Return Value

If there are no errors in the allocation of the resource, then the status `CS_SUCCESS` is returned.

6.10.1.5 Errors

The function may result in one of the following statuses:

CS_INVALID_ARG	The `BatchHandle` parameter was NULL
CS_INVALID_OPTION	The selected `Mode` was not available or the `MaxReqs` parameter value was not supported
CS_OUT_OF_RESOURCES	The function could not be completed as the library was out of resources
CS_UNSUPPORTED	The batch functions are not supported

6.10.2 csFreeBatchRequest()

Frees a batch handle previously allocated with a call to the `csAllocBatchRequest()` function.

6.10.2.1 Synopsis

CS_STATUS csFreeBatchRequest(CS_BATCH_HANDLE BatchHandle);

6.10.2.2 Parameters

IN BatchHandle	The handle previously allocated for batch requests.

6.10.2.3 Description

The `csFreeBatchRequest()` function frees all resources allocated for the requested batch handle.

6.10.2.4 Return Value

`CS_SUCCESS` is returned if there are no errors in freeing the batch resources.

6.10.2.5 Errors

The function may result in one of the following statuses:

CS_DEVICE_NOT_AVAILABLE	One or more resources referenced in the batch were not able to be accessed because the device was not available
CS_INVALID_HANDLE	The `BatchHandle` parameter was not valid
CS_NOT_DONE	One or more requests in the batch were outstanding and not completed yet
CS_UNSUPPORTED	The batch functions are not supported

6.10.3 csConfigureBatchEntry()

This function configures the batch request resource represented by the input handle based on the value of CS_BATCH_CONFIG_TYPE. The configuration is one of the following: add a new entry, delete an existing entry, reconfigure an existing entry, join two existing entries, or split off an existing entry. The CSBatchRequest structure, if specified, is one of storage, compute, or copy memory.

6.10.3.1 Synopsis

```
CS_STATUS csConfigureBatchEntry(CS_BATCH_HANDLE BatchHandle,
        CS_BATCH_CONFIG_TYPE Action, const CsBatchRequest *Req,
        CS_BATCH_INDEX Before, CS_BATCH_INDEX After,
        CS_BATCH_INDEX *Curr);
```

6.10.3.2 Parameters

IN BatchHandle	The batch request handle that describes the CSx batch to which the specified configuration action has to be applied.
IN Action	The desired configuration option on the specified `BatchHandle` parameter. The chosen option specifies the interpretation of other parameters. See Table 11 for additional details.
IN Req	This parameter, if present, specifies the batch entry to be added or reconfigured. See Table 11 for additional details.
IN Before	A valid non-zero batch entry index of a previously configured entry in the batch request that is being configured with the current `Action`. See Table 11 for additional details.
IN After	A valid non-zero batch entry index of a previously configured entry in the batch request that is being configured with the current `Action`. See Table 11 for additional details.
OUT Curr	A pointer to a buffer to hold the output of the batch entry index if a new batch entry is requested by an `Action` of `CS_BATCH_ADD`. All other `Action` options shall set this parameter to NULL. See Table 11 for additional details.

6.10.3.3 Description

The `csConfigureBatchEntry()` function performs the configuration option specified by `Action` on the batch requests specified by the `BatchHandle` parameter. This is a compound function where all parameters are required.

The `Action` parameter defines how other parameters are interpreted. The action is one of the following: add a batch entry, delete a batch entry, reconfigure an existing batch entry, split two existing batch entries, and join two existing batch entries. See 6.3.5.2.14 for details on `Action`. Table 11 describes how each parameter may be specified based on the specified `Action`.

A resulting condition is described as follows:

Non-associated entry: An entry in in a batch that is not associated with any other entries.

Head of sequence entry: The first entry in a sequence of requests that is inserted preceeding the entry denoted by `After`.

Tail of sequence entry: The last entry in a sequence of requests that is inserted following the entry denoted by `Before`.

In-between entry: An entry that is inserted between existing entries specified by the `Before` and `After` indices.

Join two entries: The entries denoted by `Before` and `After` indices are linked.

All links that follow the entry specified in Before are terminated: The entry denoted by the `Before` index becomes a Tail of sequence entry in the sequence and any entries previously connected to the output of that entry, if not connected to another Before entry becomes a Head of sequence entry.

Split all links that precede the entry specified in After: The entry denoted by the `After` index becomes a Head of sequence entry, and any entries previously connected to the input of that entry, if not connected to another After entry becomes a Tail of sequence entry.

Delete entry: The entry denoted by the `Before` index is deleted from the sequence.

Reconfigure entry: The entry denoted by the `Before` entry is reconfigured with the information provided in `Req`.

Table 11: Configure Batch Entry Actions

Action	Before	After	Req	Curr	Resulting Condition
CS_BATCH_ADD	0	0	Non-null	Non-null	Non-associated entry
CS_BATCH_ADD	0	Non-zero	Non-null	Non-null	Head of sequence entry
CS_BATCH_ADD	Non-zero	0	Non-null	Non-null	Tail of sequence entry

CS_BATCH_ADD	Non-zero	Non-zero	Non-null	Non-null	In-between entry
CS_BATCH_JOIN	Non-zero	Non-zero	Null	Null	Join two entries
CS_BATCH_SPLIT	Non-zero	0	Null	Null	All links that follow the entry specified in Before are terminated
CS_BATCH_SPLIT	0	Non-zero	Null	Null	Split all links that precede the entry specified in After
CS_BATCH_SPLIT	Non-zero	Non-zero	Null	Null	Split specific link
CS_BATCH_DELETE	Non-zero	0	Null	Null	Delete entry specified by Before
CS_BATCH_RECONFIG	Non-zero	0	Non-null	Null	Reconfigure entry specified by Before

6.10.3.4 Return Value

`CS_SUCCESS` is returned if there are no errors in processing the entry addition.

6.10.3.5 Errors

The function may result in one of the following statuses:

CS_DEVICE_NOT_AVAILABLE	One or more resources referenced in the batch were not able to be accessed as the device is not available
CS_INVALID_ARG	One or more parameters in `Req` were not valid

CS_INVALID_CSF_ID	The specified `CSFId` in the Req parameter was not a valid Id
CS_INVALID_HANDLE	The `BatchHandle` parameter was not valid, or a `DevHandle` or `MemHandle` referenced in the `Req` data structure was not valid.
CS_INVALID_OPTION	An invalid option was selected with the `Option` parameter
CS_UNKNOWN_MEMORY	The memory referenced by a memory handle in the `Req` data structure was not valid
CS_UNSUPPORTED	The batch functions are not supported

6.10.3.6 Notes

The parameter `Req` defines:

a) the individual requests themselves;
b) the type of batch request (i.e., CS_COPY_DEV_MEM, CS_STORAGE_IO, or CS_QUEUE_COMPUTE); and
c) the work item which may be one of CsCopyMemRequest, CsStorageRequest or CsComputeRequest data structures.

See details in the `csQueueCopyMemRequest()` function, the `csQueueStorageRequest()` function, and the `csQueueComputeRequest()` function.

6.10.4 csHelperResizeBatchRequest()

Resizes an existing batch request for the maximum number of requests that the batch request is able to accommodate.

6.10.4.1 Synopsis

CS_STATUS csHelperResizeBatchRequest(CS_BATCH_HANDLE BatchHandle,
int MaxReqs);

6.10.4.2 Parameters

IN BatchHandle	The handle previously allocated for batch requests that is resized.
IN MaxReqs	The maximum number of requests that this batch resource is resized to. This parameter shall not exceed the maximum supported by the CSE. This parameter shall not

be less than the current number of batch entries associated with this BatchHandle.

6.10.4.3 Description

The `csHelperResizeBatchRequest()` function resizes an existing batch request to the maximum request size specified.

6.10.4.4 Return Value

`CS_SUCCESS` is returned if there are no errors in the resizing of the resource.

6.10.4.5 Errors

The function may result in one of the following statuses:

CS_INVALID_ARG	The `BatchHandle` parameter was NULL or the `MaxReqs` parameter was less than the current number of batch entries associated with this `BatchHandle`
CS_INVALID_HANDLE	The `BatchHandle` parameter was not valid
CS_INVALID_OPTION	The `MaxReqs` parameter value was not supported
CS_OUT_OF_RESOURCES	The function could not be completed as the library was out of resources
CS_UNSUPPORTED	The batch functions are not supported

6.10.5 csQueueBatchRequest()

Queues a data graph request to the device to be executed synchronously or asynchronously in the device. The request is able to support serial, parallel or a mixed variety of batched jobs defined by their data flow and support storage, compute, and data copy requests all in one function. The handle is required to already have been populated with the list of batched requests.

6.10.5.1 Synopsis

```
CS_STATUS csQueueBatchRequest(CS_BATCH_HANDLE BatchHandle,
        void *Context, csQueueCallbackFn CallbackFn,
        CS_EVT_HANDLE EventHandle, CS_REQ_HANDLE *ReqHandle,
        u64 *CompValue);
```

6.10.5.2 Parameters

IN BatchHandle	The handle previously allocated for batch requests.
IN Context	A user specified context for the queue request when asynchronous. The parameter is required only if `CallbackFn` or `EventHandle` is specified.
IN CallbackFn	A callback function if the queue request is asynchronous.
IN EventHandle	A handle to an event previously created using the `csCreateEvent()` function. This value may be `NULL` if `CallbackFn` parameter is specified to be a valid value or if the request is required to be synchronous.
OUT ReqHandle	A pointer to a buffer to hold the request handle if successfully queued. The received handle may be used to abort this request using the `csAbortRequest()` function.This is an optional parameter and depends on the implementation.
OUT CompValue	Additional completion value provided as part of completion. This may be optional depending on the implementation.

6.10.5.3 Description

The `csQueueBatchRequest()` function queues a batch of requests that may include flows for storage, compute, and device memory copies with the CSE.

The inputs and outputs for the request are specified in the `Req` data structure which contains entries for storage, compute, and device memory copy. The details of `Req` are populated using the helper functions detailed in 6.10. The request may be performed synchronously or asynchronously. To perform the request synchronously, the parameters `CallbackFn` and `EventHandle` should be set to NULL and `Context` is ignored. To perform the request asynchronously, either a callback is required to be specified in `CallbackFn` or an event handle is required to be specified in `EventHandle`. It is an error to specify both of these parameters. If `Context` is specified, `Context` is returned in the asynchronous completion path. See notes for details.

An optional pointer may be specified to receive a `ReqHandle` for the request to allow the request to be aborted. For asynchronous operation, if a valid pointer is specified, the pointer is updated with a handle to the submitted request. A synchronous operation ignores this parameter.

For more information on callback usage, see 6.3.8.

For more information on using events for polling see 6.11.3.

6.10.5.4 Return Value

CS_SUCCESS is returned if there are no errors in synchronous queue operation.

CS_QUEUED is returned if there are no errors in asynchronous queue operation.

6.10.5.5 Errors

The function may result in one of the following statuses:

CS_DEVICE_NOT_AVAILABLE	The CSx referenced by the BatchHandle parameter was not available for processing
CS_ERROR_IN_EXECUTION	There was an error in execution of the batch request
CS_INVALID_ARG	The CallbackFn parameter and the EventHandle parameter were both set
CS_INVALID_HANDLE	The BatchHandle parameter was not valid or the EventHandle parameter was not valid
CS_UNKNOWN_MEMORY	The memory referenced by the MemHandle field in in one or more requests in the batch was not valid
CS_UNSUPPORTED	The batch functions are not supported

This function may provide additional completion value returned in CompValue.

6.10.5.6 Notes

Queueing work items in batches simplifies how a more complex operation should be done in one request. A batch of requests is able to take many forms as follows:

a) A mixture of storage operations, compute memory copy operations, and device-based CSF executions. E.g. Copy data from host memory to compute memory and run a CSF. Additionally may copy the results back to host memory;
b) Divide a large compute work item into smaller work items and run each of them on similar functions in parallel;
c) Copy multiple copies of data from device memory to host memory that may describe something similar to a scatter gather list in storage;

d) Load data from storage directly in device memory, run a CSF and copy the results back to host memory. This may be the most common type of usage;
e) Queue the output of one CSF to a subsequent CSF; and
f) Load storage data and metadata in parallel, run separate computational storage functions on them in parallel, collate the results to a secondary CSF, and copy the results back to host memory.

This is a generic queueing function to batch different operations of CSFs and device memory copy operations. It is the responsibility of the caller to ensure the number of arguments and their individual values map correctly to the CSF.

The batching operation requires that a batch handle be allocated using the `csAllocBatchRequest()` function and that individual requests be added using the `csConfigureBatchEntry()` function.

The `Mode` field input in the `csAllocBatchRequest()` function, which may be `CS_BATCH_SERIAL`, `CS_BATCH_PARALLEL`, and `CS_BATCH_HYBRID` specifies how this request is to be handled. Serialized requests are those that depend on the previous requests output as their input. Parallelized requests break down multiple requests into smaller requests that are able to be executed at the same time. Requests may be sent in parallel to the same function on the same device or different devices to be executed at the same time. For additional details on batching requests see section 5.2.1.

If the data input to a CSF has dependencies on a previous operation to complete, then the `CS_BATCH_INDEX` parameters are required to be utilized correctly to place the new request entry in the batch of requests. Each new request may be inserted anywhere in the batch and the indices help guide the queue placement. For example, a previous request may have an AFDM copy from host or a storage I/O request that is required to populate the input data to this batch request. In a serialized request using `CS_BATCH_SERIAL` mode, the storage request is placed first followed by the CSF request. The dependencies of individual requests are guided by the placement of each request in the batch list. The batch request preprocessor looks up dependencies of memory resources in the list. Optimizations on queuing requests may be applied based on this information presented by the batch details. With `CS_BATCH_HYBRID` mode, complex flow graphs are able to be processed where multiple serial and parallel flows are able to be accommodated. Additional details on this usage are provided under hybrid operations in section 5.2.1.3.

The requirements on the `CallbackFn` and `EventHandle` apply the same way as in the `csQueueCopyMemRequest()` function.An `EventHandle` is utilized only by user space applications while function space users (e.g., drivers and filesystems) uses the `CallbackFn`.

For `EventHandle`, see the `csCreateEvent()` function for usage.

The following example shows batch request processing to analyze a 1GB data file and provide the output back to the host. This example demonstrates reuse and reconfigurability.

```
// preprocess: discover & configure CSF(s), Storage
//     open file in O_DIRECT mode and locate data section
//     preallocate AFDM for inputs/outputs
// Allocate a batch request for serial mode processing
status = csAllocBatchRequest(CS_BATCH_SERIAL, 3, &BatchHandle);
if (status != CS_SUCCESS)
    ERROR_OUT("csAllocBatchRequest failed\n");
// allocate storage, compute, and DMA requests and set them up..
status = csConfigureBatchEntry(BatchHandle, CS_BATCH_ADD, &storReq, 0, 0, &storEntry);
if (status != CS_SUCCESS)
    ERROR_OUT("csConfigureBatchEntry failed for storEntry\n");
status = csConfigureBatchEntry(BatchHandle, CS_BATCH_ADD, &compReq, 0, storEntry,
&compEntry);
if (status != CS_SUCCESS)
    ERROR_OUT("csConfigureBatchEntry failed for compEntry\n");
status = csConfigureBatchEntry(BatchHandle, CS_BATCH_ADD, &copyReq, 0, compEntry,
&copyEntry);
if (status != CS_SUCCESS)
    ERROR_OUT("csConfigureBatchEntry failed for copyEntry\n");
// process through entire data file of 1GB
while (fileSize) {
    status = csQueueBatchRequest(BatchHandle, NULL, NULL, NULL, NULL, NULL);
    if (status != CS_SUCCESS)
        ERROR_OUT("csQueueBatchRequest failed\n");
    fileSize -= dataSize;
    // advance file pointer to next 1MB (only updates storage batch details)
    storReq.u.StorageIo.u.FileIo.Offset += dataSize;
    status = csConfigureBatchEntry(BatchHandle, CS_BATCH_RECONFIG,
        &storReq, storEntry, 0, NULL);
    if (status != CS_SUCCESS)
        ERROR_OUT("csConfigureBatchEntry failed\n");
}
status = csFreeBatchRequest(BatchHandle);
if (status != CS_SUCCESS)
    ERROR_OUT("csFreeBatchRequest failed\n");
```

6.11 Event Management

The following functions aid in the usage of OS abstracted events.

6.11.1 csCreateEvent()

Allocates an event resource and returns a handle when successful.

6.11.1.1 Synopsis

CS_STATUS csCreateEvent(CS_EVT_HANDLE *EventHandle);

6.11.1.2 Parameters

OUT EventHandle	Pointer to a buffer to hold the event handle once allocated

6.11.1.3 Description

The `csCreateEvent()` function allocates and initializes a system event resource.

If a valid `EventHandle` pointer is specified, the `EventHandle` is updated with the handle to the allocated event resource.

All input parameters are required for this function.

6.11.1.4 Return Value

`CS_SUCCESS` is returned if there were no errors and an event resource was successfully allocated.

6.11.1.5 Errors

The function may result in one of the following statuses:

CS_INVALID_ARG	The specified `EventHandle` pointer was NULL
CS_NOT_ENOUGH_MEMORY	There was insufficient memory to handle this request

6.11.1.6 Notes

The event resource is allocated at the system level not at the device level. That event resource is able to be used with any CSx. Once used, that event resource is referenced by that device and should not be used simultaneously by more than once device.

6.11.2 csDeleteEvent()

Frees a previously allocated event resource.

6.11.2.1 Synopsis

CS_STATUS csDeleteEvent(CS_EVT_HANDLE EventHandle);

6.11.2.2 Parameters

IN EventHandle	The event handle to be freed

6.11.2.3 Description

The `csDeleteEvent()` function deletes an event resource previously allocated using the `csCreateEvent()` function.

If a valid `EventHandle` is specified, that `EventHandle` is freed and returned to the system.

All input parameters are required for this function.

6.11.2.4 Return Value

`CS_SUCCESS` is returned if there were no errors and an event resource was successfully freed.

6.11.2.5 Errors

The function may result in one of the following statuses:

CS_HANDLE_IN_USE	The `EventHandle` was not able to be deleted as it was still in use
CS_INVALID_ARG	The `EventHandle` parameter was `NULL`
CS_INVALID_HANDLE	The `EventHandle` parameter was not valid

6.11.3 csPollEvent()

Polls the event specified for any pending events.

6.11.3.1 Synopsis

CS_STATUS csPollEvent(CS_EVT_HANDLE EventHandle, void **Context, u64 *CompValue);

6.11.3.2 Parameters

IN EventHandle	The event handle to be polled
OUT Context	The context to the event that completed
OUT CompValue	Additional completion value provided as part of completion. This may be optional depending on the implementation.

6.11.3.3 Description

The `csPollEvent()` function queries an event resource previously allocated using the `csCreateEvent()` function when used with CSFs. The `Context` parameter returned refers to the original context provided when the request was made.

If a valid `EventHandle` is specified, that `EventHandle` is queried for any pending events.

All input parameters are required for this function.

6.11.3.4 Return Value

`CS_NOT_DONE` is returned if there no pending events.

`CS_SUCCESS` is returned if the pending work item completed successfully without errors.

6.11.3.5 Errors

The function may result in one of the following statuses:

CS_DEVICE_ERROR	The device encountered an error and was not able to make progress
CS_DEVICE_NOT_AVAILABLE	The device referenced was no longer available
CS_ERROR_IN_EXECUTION	An error occurred during execution of the function
CS_FATAL_ERROR	A fatal error occurred
CS_INVALID_ARG	The `EventHandle` parameter was NULL
CS_INVALID_HANDLE	The `EventHandle` parameter was not valid
CS_IO_TIMEOUT	A timeout occurred on the event being polled
CS_NOTHING_QUEUED	There was nothing queued for this `EventHandle`

This function may provide additional completion value returned in `CompValue`.

6.11.3.6 Notes

An event resource is submitted to the `csQueueCopyMemRequest()` function, the `csQueueComputeRequest()` function, or the `csQueueBatchRequest()` function for polling. It is the responsibility of the user to ensure that the correct event handle is used to poll and that the handle had not been freed use the `csDeleteEvent()` function.

6.12 Management

Device management provides functions that are used to query and manage the device:

a. properties; and
b. resources.

6.12.1 csQueryDeviceProperties()

Queries the CSx for its properties.

This is a privileged function.

6.12.1.1 Synopsis

```
CS_STATUS csQueryDeviceProperties(CS_DEV_HANDLE DevHandle,
        CS_RESOURCE_TYPE Type, int *Length, CsProperties *Buffer);
```

6.12.1.2 Parameters

IN DevHandle	Handle to CSx
IN Type	The type of CSx resource to query
IN OUT Length	A pointer to a buffer that holds or is able to hold the length of `Buffer`
OUT Buffer	A pointer to a buffer to hold the device properties

6.12.1.3 Description

The `csQueryDeviceProperties()` function fills `Buffer` with the device properties for the CSx as requested by the `Type` field, if the length specified in `Length` is sufficient. This function, if successful, may return one or more sub-structures in `Buffer`.

If a valid `Buffer` pointer is provided, where the length specified in `Length` is sufficient, then the `Buffer` is updated with the requested CSx resource type properties and `Length` is updated with the total data returned in bytes in `Buffer`. If the length specified in `Length` is not sufficient to hold the contents returned in `Buffer`, then `Length` is populated with the required buffer size and a status of CS_INVALID_LENGTH is returned.

If a NULL pointer is specified for Buffer and a valid pointer is provided for Length, then the required buffer size is returned in Length for that resource type. The user should allocate a buffer of the returned size and reissue the request.

If a valid pointer is specified for Buffer, a valid pointer is provided for Length, and the value in Length is not sufficient for the device properties, then the required buffer size is returned in Length.

All input parameters are required for this function.

6.12.1.4 Return Value

This function returns:

a) CS_SUCCESS if there is no error and zero or more device properties are returned; or

b) CS_LENGTH_RETURNED if NULL is specified for Buffer and a valid pointer is provided for Length and Length is returned.

6.12.1.5 Errors

The function may result in one of the following statuses:

CS_DEVICE_ERROR	The device encountered and error and was not able to make progress
CS_DEVICE_NOT_AVAILABLE	The device specified by the DevHandle parameter was not available
CS_DEVICE_NOT_READY	The device specified by the DevHandle parameter was not ready
CS_INVALID_ARG	The DevHandle parameter was NULL or the Length was NULL
CS_INVALID_HANDLE	The DevHandle parameter was not valid
CS_INVALID_ID	The specified resource in Type was not valid
CS_INVALID_LENGTH	The size of Buffer provided in the Length parameter was not sufficient
CS_NO_PERMISSIONS	There were insufficient user permissions to satisfy this request

CS_NOT_ENOUGH_MEMORY	There was insufficient memory to satisfy this request

6.12.1.6 Notes

The properties returned provide information on versions in use and are able to be used by the caller when multiple devices are in use.

A user utilizes this function early on in device setup to verify that the properties are as expected prior to configuring the CSx.

6.12.2 csQueryDeviceStatistics()

Queries the CSx for specific runtime statistics. These could vary depending on the requested type inputs. Details on CSFs and the CSx may be queried.

This is a privileged function.

6.12.2.1 Synopsis

CS_STATUS csQueryDeviceStatistics(CS_DEV_HANDLE DevHandle,
CS_STAT_TYPE Type, void *Identifier, CsStatsInfo *Stats);

6.12.2.2 Parameters

IN DevHandle	Handle to CSx
IN Type	Statistics type to query
IN Identifier	Additional options based on Type
OUT Stats	A pointer to a buffer to hold the requested statistics

6.12.2.3 Description

The `csQueryDeviceStatistics()` function returns the device statistics based on `Type` requested. The `Stats` field is a union of structures and is populated with the desired output based on the input provided by `Type` and `Identifier` fields.

The `identifier` parameter is optional and is required only for certain statistics types. The `Identifier` parameter is used with structures `CSEDetails` and `CSFDetails`. When used for `CSEDetails`, the `Identifier` parameter refers to the `CSEId` field in `CSEProperties`. When used for `CSFDetails`, the `Identifier` refers to the `CSFId` statistics to be queried.

For a specific CSE's statistics, the `Identifier` parameter should be set to its unique `CSEId` available in the `csQueryDeviceProperties()` function.Similarly, for specific CSF statistics, the Identifier is required to be set to its unique `CSFId` also available using the `csQueryDeviceProperties()` function. An error is returned if the `Identifier` parameter is set to NULL and `Type` requires a valid `Identifier`.

All input parameters are required for this function.

6.12.2.4 Return Value

`CS_SUCCESS` is returned if there are no errors.

6.12.2.5 Errors

The function may result in one of the following statuses:

CS_DEVICE_ERROR	The device encountered an error and was not able to make progress
CS_DEVICE_NOT_AVAILABLE	The device specified by the `DevHandle` parameter was not available
CS_DEVICE_NOT_READY	The device specified by the `DevHandle` parameter was not ready
CS_INVALID_ARG	The `DevHandle` parameter was NULL, `Type` was zero, or `Stats` was NULL
CS_INVALID_HANDLE	The `DevHandle` parameter was not valid
CS_NO_PERMISSIONS	There were insufficient user permissions to satisfy this request
CS_NOT_ENOUGH_MEMORY	There was insufficient memory to satisfy this request

6.12.2.6 Notes

The Statistics returned provide information on CSx usage (e.g., utilization and health). Some of the statistics reflected are preserved since the power on state. The counters are not reset on a query.

6.12.3 csCSFDownload()

Downloads a specified CSF resource. It is implementation specific as to how the downloaded code is secured.

This is a privileged function.

6.12.3.1 Synopsis

CS_STATUS csCSFDownload(CS_DEV_HANDLE DevHandle,
    const CsCSFDownloadInfo *Info, u32 *CSFId);

6.12.3.2 Parameters

IN DevHandle	Handle to CSx
IN Info	A pointer to a buffer that holds the CSF resource details to download
OUT CSFId	A pointer to a buffer to hold the identifier to the downloaded CSF resource

6.12.3.3 Description

The `csCSFDownload()` function downloads a CSF using the details in `Info`. The `Info` parameter provides the details of download contents such as the CS_CSF_RESOURCE_TYPE. Additional details on these fields are provided in section 6.3.5.3.13 and 6.3.5.2.4. On a successful download, a `CSFId` for the downloaded CSF is returned. This value may be used to configure the downloaded resource using the `csConfig()` function.

All parameters are required for this function.

6.12.3.4 Return Value

`CS_SUCCESS` is returned if there are no errors.

6.12.3.5 Errors

The function may result in one of the following statuses:

CS_DEVICE_NOT_AVAILABLE	The device specified by the `DevHandle` parameter was not available
CS_INVALID_ARG	The `DevHandle` parameter was NULL, `Info` was NULL, or `CSFId` was NULL
CS_INVALID_HANDLE	The `DevHandle` parameter was not valid

CS_INVALID_OPTION	An option selected in `Info` was not valid
CS_NO_PERMISSIONS	There were insufficient user permissions to satisfy this request
CS_UNSUPPORTED	This function is not supported
CS_UNSUPPORTED_INDEX	The specified index in `Info` is not supported
CS_UNSUPPORTED_TYPE	The specified type in `Info` is not supported

6.12.3.6 Notes

CSxes that contain a CSEE that is not capable of accepting a downloaded CSF fail this function (e.g., CSx devices that only have fixed functionality).

6.12.4 csConfig()

Configures the activation state or vendor specific configuration of the specified CSx. The CSEE and CSF are the resources that may be activated or configured with this function. Prior to usage, these resources are required to be activated.

This is a privileged function.

6.12.4.1 Synopsis

```
CS_STATUS csConfig(CS_DEV_HANDLE DevHandle, int *Length,
        const CsConfigInfo *Info, CsConfigData *Data);
```

6.12.4.2 Parameters

IN DevHandle	Handle to CSx
IN Length	Length of Info when vendor configuration is specified
IN Info	A pointer to a buffer that holds the data structure with the requested configuration
OUT Data	Configuration results

6.12.4.3 Description

The csConfig() function configures the specified CSx resource. The requested configuration is specified in Info and the results of the configuration are provided as output in Data. The Length parameter is specified when implementation specific details are described in the VSInfo field in the Info parameter.

The Length parameter is optional based on the presence of VSInfo. All other parameters are required for this function.

6.12.4.4 Return Value

CS_SUCCESS is returned if there are no errors.

6.12.4.5 Errors

The function may result in one of the following statuses:

CS_DEVICE_ERROR	The device encountered an error and was not able to make progress
CS_DEVICE_NOT_AVAILABLE	The device specified by the DevHandle parameter was not available
CS_INVALID_ARG	The DevHandle parameter was NULL, Info was NULL, or Data was NULL
CS_INVALID_HANDLE	The DevHandle parameter was not valid
CS_INVALID_ID	The specified Id in Info structure was not valid
CS_INVALID_LENGTH	The size of Buffer provided in the Length parameter was not sufficient
CS_INVALID_OPTION	A valid option was not selected
CS_LOAD_ERROR	The specified configuration was not able to be loaded
CS_NO_PERMISSIONS	There were insufficient user permissions to satisfy this request
CS_NOT_ENOUGH_MEMORY	There was not enough memory to satisfy this request
CS_UNSUPPORTED	This function is not supported

6.12.4.6 Notes

This function is only accepted by CSxes that contain a CSE that is capable of processing configuration input.

6.12.5 csReset()

Resets the CSx resource specified.

This is a privileged function.

6.12.5.1 Synopsis

CS_STATUS csReset(CS_DEV_HANDLE DevHandle,
    CS_RESOURCE_TYPE ResourceType, u32 ResourceId);

6.12.5.2 Parameters

IN DevHandle	Handle to CSx
IN ResourceType	Type of resource to reset
IN ResourceId	The Identifier of the resource to reset

6.12.5.3 Description

The `csReset()` function resets the specified CSx resource . As part of the operation, outstanding transactions to one or more of the CSEs are aborted and IOs are de-queued and completed in error.

All input parameters are required for this function.

6.12.5.4 Return Value

`CS_SUCCESS` is returned if there are no errors.

6.12.5.5 Errors

The function may result in one of the following statuses:

CS_DEVICE_ERROR	The device encountered an error and was not able to make progress
CS_DEVICE_NOT_AVAILABLE	The device specified by the `DevHandle` parameter was not available
CS_DEVICE_NOT_READY	The device specified by the `DevHandle` parameter was not ready

CS_INVALID_ARG	The DevHandle parameter was NULL, ResourceType was NULL, or ResourceId was zero
CS_FATAL_ERROR	A fatal error occurred and the request was not able to be aborted
CS_INVALID_HANDLE	The DevHandle parameter was not valid
CS_INVALID_ID	The specified ResourceId was not valid
CS_NO_PERMISSIONS	There were insufficient user permissions to satisfy this request
CS_NOT_ENOUGH_MEMORY	There was not enough memory to satisfy this request
CS_UNSUPPORTED	This function is not supported

6.12.5.6 Notes

The call is only able to be done by a privileged user.

6.13 Library management

Library management involves functions that are used to query and manage the CS API library interfaces and resources for compute offload devices. These library functions may be used to add additional functionality not available in the CS API library, achieve compatibility, or to enable vendor specific requirements.

6.13.1 csQueryLibrarySupport()

Queries the CS API library for supported functionality. Any application that uses the CS API library is able to use this query.

6.13.1.1 Synopsis

CS_STATUS csQueryLibrarySupport(CS_LIBRARY_SUPPORT Type,
int *Length, char *Buffer);

6.13.1.2 Parameters

IN Type	Library support type query
IN OUT Length	A pointer to a buffer that holds or is able to hold the length of Buffer

OUT Buffer	Returns a list of queried items

6.13.1.3 Description

The csQueryLibrarySupport() function fills Buffer with a list of all items for query based on Type, if the length specified in Length is sufficient. The output copied to Buffer is a set of strings separated by commas.

If a valid Buffer pointer is specified where the length specified in Length is sufficient, then the buffer is updated with the list of all items that match support for Type to the length of the string. If the length specified in Length is not sufficient to hold the contents returned in Buffer, then Length is populated with the required buffer size and a status of CS_INVALID_LENGTH is returned.

If a NULL pointer is specified for Buffer and a valid pointer is provided for Length, then the required buffer size is returned in Length. The user should allocate a buffer of the returned size and reissue the request. The user may also provide a large enough buffer and satisfy the request.

All input and output parameters are required for this function.

6.13.1.4 Return Value

This function returns:

a) CS_SUCCESS if there is no error and zero or more items are returnedfor the query; or
b) CS_LENGTH_RETURNED if NULL is specified for Buffer and a valid pointer is provided for Length and Length is returned.CS_SUCCESS if there is no error and the query for Type was met.

6.13.1.5 Errors

The function may result in one of the following statuses:

CS_INVALID_ARG	The Type parameter was NULL or Length was NULL
CS_INVALID_LENGTH	The size of Buffer provided in the Length parameter was not sufficient

6.13.1.6 Notes

The caller should always check the value of `Length` for a non-zero value, which represents valid entries in `Buffer` for the specified query. A null terminated string is returned in `Buffer` when `Length` is non-zero. This function may still return success when `Length` is zero.

The returned queried list is able to be parsed and verified as the user intended.

A typical source fragment implementation to return file system support would be

```
length = 0;
status = csQueryLibrarySupport(CS_FILE_SYSTEMS_SUPPORTED, &length, NULL);
if (status != CS_INVALID_LENGTH)
    ERROR_OUT("csQueryLibrarySupport returned unknown error\n");

fs_list = malloc(length);
status = csQueryLibrarySupport(CS_FILE_SYSTEMS_SUPPORTED, &length, &fs_list[0]);
if (status != CS_SUCCESS)
    ERROR_OUT("csQueryLibrarySupport returned error\n);
// verify if XFS filesystem is supported
…
```

6.13.2 csRegisterPlugin()

Registers a specified plugin with the CS API library.

This is a privileged function.

6.13.2.1 Synopsis

CS_STATUS csRegisterPlugin(const CsPluginRequest *Req);

6.13.2.2 Parameters

IN Req	Request structure to register a plugin

6.13.2.3 Description

The `csRegisterPlugin()` function registers the specified plugin.

All input parameters are required for this function.

6.13.2.4 Return Value

`CS_SUCCESS` is returned if there are no errors.

6.13.2.5 Errors

The function may result in one of the following statuses:

CS_INVALID_ARG	The Req parameter was NULL
CS_INVALID_OPTION	The option selected in the Req structure was not valid
CS_NO_PERMISSION	There were insufficient user permissions to satisfy this request
CS_NOT_ENOUGH_MEMORY	There was not enough memory to satisfy this request

6.13.2.6 Notes

This functionality is used by a privileged process to register a plugin in the system. Computational Storage Device providers and vendors who provide their own plugin support would use this function.

6.13.3 csDeregisterPlugin()

Deregisters a specified plugin from the CS API library.

This is a privileged function.

6.13.3.1 Synopsis

CS_STATUS csDeregisterPlugin(const CsPluginRequest *Req);

6.13.3.2 Parameters

IN Req	Request structure to deregister a plugin

6.13.3.3 Description

The `csDeregisterPlugin()` function deregisters the specified plugin.

All input parameters are required for this function.

6.13.3.4 Return Value

`CS_SUCCESS` is returned if there are no errors.

6.13.3.5 Errors

The function may result in one of the following statuses:

CS_INVALID_ARG	The `Req` parameter was NULL
CS_INVALID_OPTION	The option selected in the `Req` structure was not valid
CS_NO_PERMISSIONS	There were insufficient user permissions to satisfy this request
CS_NOT_ENOUGH_MEMORY	There was not enough memory to satisfy this request

6.13.3.6 Notes

This functionality is used by a privileged process to deregister a plugin in the system. Computational Storage Device providers and vendors who provide their own plugin support use this function.

A Sample Code

A.1 Initialization and queuing a synchronous request

A synchronous (blocking) request where the user waits for the I/O to complete is illustrated in the following decryption example which exercises the following steps

a) Discover the CSx and access it;

b) Discover the CSF to run decryption;

c) Allocate device memory;

d) Transfer encrypted data from host memory to device; and

e) Execute the CSF.

Initialization may occur in the following way:

```
// discover my device
length = sizeof(csxBuffer);
status = csGetCSxFromPath("myFileToAccelerate", &length, &csxBuffer);
if (status != CS_SUCCESS)
    ERROR_OUT("No CSx device found!\n");
// open device, init function, and prealloc buffers
status = csOpenCSx(csxBuffer, &MyDevContext, &dev);
if (status != CS_SUCCESS)
    ERROR_OUT("Could not access device\n");

// query run details of decrypt CSF
status = csGetCSFId(dev, "decrypt", &infoLength, &count, &csfInfo);
if (status != CS_SUCCESS)
    ERROR_OUT("CSX does not contain any decrypt CSFs \n");
// pick highest performant CSF from returned list
CSFIdInfo *p = csfInfo;
CSFIdInfo *myCSF = NULL;
for (i=0; i< count; i++, p++) {
    if ((myCSF == NULL) ||
         ((myCSF != NULL) && (p->RelativePerformance > myCSF->RelativePerformance))) {
         myCSF = p;
    }
}
decryptId = myCSF->CSFId;


// Next, pick the most performant FDM for chosen CSF
FDMAccess *p = myCSF->FDMList;
FDMAccess *myFDM = NULL;
for (i = 0; i < myCSF->NumFDMs; i++, p++) {
```

```
    if ((myFDM == NULL) ||
        ((myFDM != NULL) && (p->RelativePerformance > myFDM->RelativePerformance))) {
        myFDM = p;
    }
}
// allocate device memory
CsMemFlags f;
f.s->FDMId = myFDM->FDMId;
f.s->Flags = 0;   // may also be CS_FDM_CLEAR
for (i = 0; i < 2; i++) {
    status = csAllocMem(dev, CHUNK_SIZE, &f, 0, &AFDMArray[i] , NULL);
    if (status != CS_SUCCESS)
        ERROR_OUT("AFDM alloc error\n");
}
```

Source data may be fetched in the following way:

```
// next, copy encrypted data from host memory into AFDM
// allocate copy request and issue it
CsCopyMemRequest copyReq = malloc(sizeof(CsCopyMemRequest));
if (!copyReq)
    ERROR_OUT("request alloc error\n");
// setup copy request
copyReq->Type = CS_COPY_TO_DEVICE;
copyReq->u.HostVAddress = encrypt_buf;
copyReq->DevMem.MemHandle = AFDMArray[0];
copyReq->DevMem.ByteOffset = 0;
copyReq->Bytes = CHUNK_SIZE;
// issue a synchronous copy request
status = csQueueCopyMemRequest(copyReq, copyReq, NULL, NULL, NULL, NULL);
if (status != CS_SUCCESS)
    ERROR_OUT("Copy to AFDM error\n");
```

Compute execution may be performed in the following way.

```
// allocate compute request for 3 args
CsComputeRequest compReq = malloc(sizeof(CsComputeRequest) + \
    (sizeof(CsComputeArg) * 3));
if (!compReq)
    ERROR_OUT("request alloc error\n");
// setup work request
compReq->CSFId = decryptId;
compReq->NumArgs = 3;
argPtr = &compReq->Args[0];
csHelperSetComputeArg(&argPtr[0], CS_AFDM_TYPE, AFDMArray[0], 0);
csHelperSetComputeArg(&argPtr[1], CS_32BIT_VALUE_TYPE, CHUNK_SIZE);
csHelperSetComputeArg(&argPtr[2], CS_AFDM_TYPE, AFDMArray[1], 0);
// issue a synchronous compute request
status = csQueueComputeRequest(compReq, compReq, NULL, NULL, NULL, NULL);
if (status != CS_SUCCESS)
    ERROR_OUT("Compute exec error\n");
```

A.2 Queuing an asynchronous request

The above example is able to be modified to be an asynchronous non-blocking request for compute offload. There are 2 asynchronous mechanisms: event based and callback based.

The following code snippet demonstrates the changes to compute execution while applying an event based mechanism.

```
// allocate event for async processing
status = csCreateEvent(&evtHandle);
if (status != CS_SUCCESS)
       ERROR_OUT("Could not create event\n");
// allocate compute request for 3 args
CsComputeRequest compReq = malloc(sizeof(CsComputeRequest) + (sizeof(CsComputeArg) *
3));
if (!compReq)
    ERROR_OUT("request alloc error\n");
// setup work request
compReq->CSFId = decryptId;
compReq->NumArgs = 3;
argPtr = &compReq->Args[0];
csHelperSetComputeArg(&argPtr[0], CS_AFDM_TYPE, AFDMArray[0], 0);
csHelperSetComputeArg(&argPtr[1], CS_32BIT_VALUE_TYPE, CHUNK_SIZE);
csHelperSetComputeArg(&argPtr[2], CS_AFDM_TYPE, AFDMArray[1], 0);
// issue an event based asynchronous compute request
status = csQueueComputeRequest(compReq, compReq, NULL, evtHandle, NULL, NULL);
if (status != CS_SUCCESS)
       ERROR_OUT("Compute exec error\n");
while ((status = csPollEvent(evtHandle, &context, NULL)) != CS_SUCCESS) {
       // I/O not done; do other work
}
```

If the event usage is swapped with a callback based model, the sample code changes as follows. No event creation is required.

```
// issue a callback based asynchronous compute request
status = csQueueComputeRequest(compReq, compReq, MyAsyncCbFn, NULL, NULL, NULL);
if (status != CS_SUCCESS)
       ERROR_OUT("Compute exec error\n");
// do other work till callback MyAsyncCbFn is invoked in separate thread context
```

A.3 Using batch processing

Batch processing aids in processing of more than one request optimally as one `csQueueBatchRequest()` function is able to take multiple requests as process them as a single request (see section 6.10). The following example illustrates a sequence of serialized batch processing requests. In the first request, data is first read from the storage device and populated in the first AFDM buffer. In the second request, the CSF is executed on the data read to decompress its contents into a second AFDM buffer. In the third request, the contents of the second buffer are copied into host memory. The batch of requests are set to execute serially and are dependent on the serialization for the final output which is handled by this batch type. The request is set to execute asynchronously in non-blocking mode.

```
// batch execute storage I/O + compute offload + DMA results to host
//
// allocate a batch request handle
status = csAllocBatchRequest(CS_BATCH_SERIAL, 3, &batchHandle);
if (status != CS_SUCCESS)
       ERROR_OUT("batch request allocation error\n");
// setup storage I/O. Batch only for LBA based I/O
```

```
// for others use normal file I/O not with batch
storReq = malloc(sizeof(CsBatchRequest));
if (!storReq)
       ERROR_OUT("memory alloc error\n");
storReq->reqType = CS_STORAGE_IO;
storReq->u.StorageIo.Mode = CS_STORAGE_BLOCK_IO;
storReq->u.StorageIo.StorageIndex = 0;
storReq->u.StorageIo.DevHandle = devHandle;
storReq->u.StorageIo.u.BlockIo.Type = CS_STORAGE_LOAD_TYPE;
storReq->u.StorageIo.u.BlockIo.DevMem.MemHandle = inMemHandle;
storReq->u.StorageIo.u.BlockIo.DevMem.ByteOffset = 0;
storReq->u.StorageIo.u.BlockIo.NumRanges = 1;
storReq->u.StorageIo.u.BlockIo.Range[0].NamespaceId = NSId;
storReq->u.StorageIo.u.BlockIo.Range[0].StartLba = LBAs[0];
storReq->u.StorageIo.u.BlockIo. Range[0].NumBlocks = 1;
status = csConfigureBatchEntry(batchHandle, CS_BATCH_ADD, storReq, 0, 0, &storEntry);
if (status != CS_SUCCESS)
       ERROR_OUT("batch request error\n");
// next, setup compute I/O with 3 CSF arguments
compReq = malloc(sizeof(CsBatchRequest) + (sizeof(CsComputeArg) * 3));
if (!compReq)
       ERROR_OUT("memory alloc error\n");
compReq->reqType = CS_QUEUE_COMPUTE;
compReq->u.Compute. CSFId = funcId;
compReq->u.Compute.NumArgs = 3;
argPtr = &compReq->u.Compute.Args[0];
csHelperSetComputeArg(&argPtr[0], CS_AFDM_TYPE, inMemHandle, 0);
csHelperSetComputeArg(&argPtr[1], CS_32BIT_VALUE_TYPE, 4096 * 3);
csHelperSetComputeArg(&argPtr[2], CS_AFDM_TYPE, outMemHandle, 0);
status = csConfigureBatchEntry(batchHandle, CS_BATCH_ADD, compReq, 0, storEntry,
                           &compEntry);
if (status != CS_SUCCESS)
       ERROR_OUT("batch request error\n");
// lastly, setup DMA results to host
copyReq = malloc(sizeof(CsBatchRequest));
if (!copyReq)
       ERROR_OUT("memory alloc error\n");
copyReq->reqType = CS_COPY_DEV_MEM;
copyReq->u.CopyMem.Type = CS_COPY_FROM_DEVICE;
copyReq->u.CopyMem.u.HostVAddress = resBuffer;
copyReq->u.CopyMem.DevMem.MemHandle = outMemHandle;
copyReq->u.CopyMem.DevMem.ByteOffset = 0;
copyReq->u.CopyMem.Bytes = 4096 * 3;
status = csConfigureBatchEntry(batchHandle, CS_BATCH_ADD, copyReq, 0, compEntry,
                           &copyEntry);
if (status != CS_SUCCESS)
       ERROR_OUT("batch request error\n");
// now queue batch request
status = csQueueBatchRequest(batchHandle, NULL, NULL, evtHandle, NULL, NULL);
if (status != CS_SUCCESS)
       ERROR_OUT("Batch exec error\n");
while ((status = csPollEvent(evtHandle, &context, NULL)) != CS_SUCCESS) {
       // I/O not done; do other work
}
```

A.4 Applying hybrid batch processing feature

The following example demonstrates how to use dependency in batch requests to create a hybrid processing model, where an input completion is required prior to starting the next request. The example reads data from storage, runs parallel compute offload operation on the data, and once complete, copies the results scattered in device memory back to host memory buffer. The example is able to be representative of analytical data that is read and computed on, and whose results are collated and provided back to host. In this example, 128KB of data is read and 32KB of results are collected.

```
// hybrid batch setup execution
// large storage I/O + 8 parallel compute requests + 8 parallel copy results to host
//
// allocate enough resources for batch request handle
status = csAllocBatchRequest(CS_BATCH_HYBRID, 1 + 8 + 8, &batchHandle);
if (status != CS_SUCCESS)
       ERROR_OUT("batch request allocation error\n");
// setup storage I/O. Batch only LBA based I/O
// for others use normal file I/O not with batch
storReq = malloc(sizeof(CsBatchRequest));
if (!storReq)
       ERROR_OUT("memory alloc error\n");
// read 128kb data from Storage into device memory
storReq->reqType = CS_STORAGE_IO;
storReq->u.StorageIo.Mode = CS_STORAGE_BLOCK_IO;
storReq->u.StorageIo.DevHandle = devHandle;
storReq->u.StorageIo.u.BlockIo.Type = CS_STORAGE_LOAD_TYPE;
storReq->u.StorageIo.u.BlockIo.StorageIndex = 0;
storReq->u.StorageIo.u.BlockIo.DevMem.MemHandle = inMemHandle;
storReq->u.StorageIo.u.BlockIo.DevMem.ByteOffset = 0;
storReq->u.StorageIo.u.BlockIo.NumRanges = 1;
storReq->u.StorageIo.u.BlockIo.Range[0].NamespaceId = NSId;
storReq->u.StorageIo.u.BlockIo.Range[0].StartLba = LBAs[0];
storReq->u.StorageIo.u.BlockIo.Range[0].NumBlocks = 32;
status = csConfigureBatchEntry(batchHandle, CS_BATCH_ADD, storReq, 0, 0, &storEntry);
if (status != CS_SUCCESS)
       ERROR_OUT("batch request error\n");
// allocate memory for parallel compute batch requests and reuse req
compReq = malloc(sizeof(CsBatchRequest) + (sizeof(CsComputeArg) * 3));
if (!compReq)
       ERROR_OUT("memory alloc error\n");
inMemOffset = 0;
for (i = 0; i < 8; i++) {
       // next, setup compute I/O with 3 arguments each
       compReq->reqType = CS_QUEUE_COMPUTE;
       compReq->u.Compute.CSFId = csfId;
       compReq->u.Compute.NumArgs = 3;
       argPtr = &compReq->u.Compute.Args[0];
csHelperSetComputeArg(&argPtr[0], CS_AFDM_TYPE, inMemHandle, inMemOffset);
csHelperSetComputeArg(&argPtr[1], CS_32BIT_VALUE_TYPE, 16384);
csHelperSetComputeArg(&argPtr[2], CS_AFDM_TYPE, outMemArray[i], 0);
       status = csConfigureBatchEntry(batchHandle, CS_BATCH_ADD, compReq, 0,
                                  storEntry, &computeEntryArray[i]);
       if (status != CS_SUCCESS)
              ERROR_OUT("batch request error\n");
       // distribute source buffer sequentially
       inMemOffset += 16384;
}
// now allocate memory for parallel DMA batch requests and reuse req
copyReq = malloc(sizeof(CsBatchRequest));
```

```
if (!copyReq)
       ERROR_OUT("memory alloc error\n");
outMemOffset = 0;
for (j = 0; j < 8; j++) {
       // lastly setup DMA results to host at 4kb offsets
       copyReq->reqType = CS_COPY_DEV_MEM;
       copyReq->u.CopyMem.Type = CS_COPY_FROM_DEVICE;
       copyReq->u.CopyMem.u.HostVAddress = &resBuffer[outMemOffset];
       copyReq->u.CopyMem.DevMem.MemHandle = outMemArray[j];
       copyReq->u.CopyMem.DevMem.ByteOffset = 0;
       copyReq->u.CopyMem.Bytes = 4096;
       status = csConfigureBatchEntry(batchHandle, CS_BATCH_ADD, copyReq, 0,
                                computeEntryArray[j], &copyEntryArray[j]);
       if (status != CS_SUCCESS)
              ERROR_OUT("batch request error\n");
       // increment destination host buffer sequentially for one final output
       outMemOffset += 4096;
}
// all done, queue the batch request
status = csQueueBatchRequest(batchHandle, NULL, NULL, evtHandle, NULL, NULL);
if (status != CS_SUCCESS)
       ERROR_OUT("batch request error\n");
// wait on the final results
while ((status = csPollEvent(evtHandle, &context, NULL)) != CS_SUCCESS) {
       // I/O not done; do other work
       // poll for previous IOs too and mark them done
}
```

A.5 Using files for storage I/O

Using the filesystem managed files for reading and writing data is a powerful interface that the `csQueueStorageRequest()` function provides. The following example demonstrates using a file to read data at a particular offset and provide those contents to a CSF.

Files used by the CS API library are required to be opened using the `O_DIRECT` flag. The file handle returned by the operating system is able to then be utilized by the function as shown below. 128K bytes are read from storage using the file handle and loaded in AFDM. Data read or written by this method are required to follow block granularity and alignment guidelines for the `Offset` and `Bytes` fields or the call may fail.

```
// query capabilities for file I/O in API library
status = csQueryLibrarySupport(CS_FILE_SYSTEMS_SUPPORTED, &buflen, &buf);
if (status != CS_SUCCESS)
       ERROR_OUT("Could not query device properties\n");
// verify if filesystem is supported

// allocate storage I/O request for file usage
storReq = malloc(sizeof(CsStorageRequest));
if (!storReq)
       ERROR_OUT("memory alloc error\n");

// setup request to read 128kb from the start of the file
storReq->Mode = CS_STORAGE_FILE_IO;
storeReq->DevHandle = devHandle;
storReq->u.CsFileIo.Type = CS_STORAGE_LOAD_TYPE;
```

```
storReq->u.CsFileIo.FileHandle = fd;
storReq->u.CsFileIo.Offset = 0;
storReq->u.CsFileIo.Bytes = 128 * 1024;
storReq->u.CsFileIo.DevMem.MemHandle = inMemHandle;
storReq->u.CsFileIo.DevMem.ByteOffset = 0;
status = csQueueStorageRequest(storReq, storReq, NULL, evtHandle, NULL, NULL);
if (status != CS_SUCCESS)
	ERROR_OUT("Storage request error\n");

// wait on the request to complete or do some other work
while ((status = csPollEvent(evtHandle, &context)) != CS_SUCCESS) {
	// I/O not done; do other work
}
```