Computational Storage API
Version 0.8 rev 0

ABSTRACT:
This SNIA Draft Standard defines the interface between an application and a Computational Storage device (CSx). For each CSx there will need to be a library that performs the mapping from the APIs in this specification and the CSx on the specific interface for that CSx.

Publication of this Working Draft for review and comment has been approved by the Computational Storage TWG. This draft represents a “best effort” attempt by the Computational Storage TWG to reach preliminary consensus, and it may be updated, replaced, or made obsolete at any time. This document should not be used as reference material or cited as other than a “work in progress.” Suggestions for revisions should be directed to http://www.snia.org/feedback/.

Working Draft
June 29, 2022
USAGE

Copyright © 2022 SNIA. All rights reserved. All other trademarks or registered trademarks are the property of their respective owners.

The SNIA hereby grants permission for individuals to use this document for personal use only, and for corporations and other business entities to use this document for internal use only (including internal copying, distribution, and display) provided that:

1. Any text, diagram, chart, table or definition reproduced shall be reproduced in its entirety with no alteration, and,

2. Any document, printed or electronic, in which material from this document (or any portion hereof) is reproduced, shall acknowledge the SNIA copyright on that material, and shall credit the SNIA for granting permission for its reuse.

Other than as explicitly provided above, you may not make any commercial use of this document or any portion thereof, or distribute this document to third parties. All rights not explicitly granted are expressly reserved to SNIA.

Permission to use this document for purposes other than those enumerated above may be requested by e-mailing tcmd@snia.org. Please include the identity of the requesting individual and/or company and a brief description of the purpose, nature, and scope of the requested use.

All code fragments, scripts, data tables, and sample code in this SNIA document are made available under the following license:

BSD 3-Clause Software License

Copyright (c) 2022, The Storage Networking Industry Association.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

* Neither the name of The Storage Networking Industry Association (SNIA) nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
DISCLAIMER

The information contained in this publication is subject to change without notice. The 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. The 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.
## Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Sections</th>
<th>Originator:</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0 r0</td>
<td>12/10/2020</td>
<td>All</td>
<td>Bill Martin</td>
<td>Initial draft – incorporation of API proposal 0.1r2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Oscar Pinto</td>
<td></td>
</tr>
<tr>
<td>0.0 r0a</td>
<td>12/11/2020</td>
<td>6.6.2, 6.12.8, 6.12.9, 6.13</td>
<td>Bill Martin</td>
<td>Minor editorial fixes to font colors</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Oscar Pinto</td>
<td></td>
</tr>
<tr>
<td>0.0 r1</td>
<td>1/22/2021</td>
<td>4</td>
<td>Bill Martin</td>
<td>Updated document per USL subgroup meeting requests from RFC document</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0 r2</td>
<td>2/2/2021</td>
<td>1 – 6.4.1</td>
<td>Bill Martin</td>
<td>Updates from 1/26 subgroup call</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Added reference to architecture document</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Added definition of CSE name</td>
</tr>
<tr>
<td>0.0 r3</td>
<td>3/16/2021</td>
<td></td>
<td>Bill Martin</td>
<td>Included Oscar Pinto proposed additions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Incorporated numerous changes from Intel comments resolved by 2/23/2021 meeting</td>
</tr>
<tr>
<td>0.0 r4</td>
<td>3/23/2021</td>
<td></td>
<td>Bill Martin</td>
<td>Added corrections to Annex A that were not included from the Oscar Pinto proposed additions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Added all unresolved Intel comments</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Other editorial fixes</td>
</tr>
<tr>
<td>0.0 r5</td>
<td>3/23/2021</td>
<td></td>
<td>Bill Martin</td>
<td>Notes and changes from User Space Library meeting</td>
</tr>
<tr>
<td>0.0 r6</td>
<td>3/23/2021</td>
<td></td>
<td>Bill Martin</td>
<td>Clean version with all comments resolved prior to 3/23/2021 meeting removed and all change marking removed</td>
</tr>
<tr>
<td>0.0 r7</td>
<td>4/23/2021</td>
<td></td>
<td>Bill Martin</td>
<td>All comments recorded and resolution included where applicable</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>No other changes to document</td>
</tr>
<tr>
<td>0.0 r8</td>
<td>4/23/2021</td>
<td></td>
<td>Bill Martin</td>
<td>Accepted all changes from 0.0r7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Added Query Properties (Oscar Pinto)</td>
</tr>
<tr>
<td>Revision</td>
<td>Date</td>
<td>Author(s)</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>--------------</td>
<td>---------------------</td>
<td>-------------</td>
<td></td>
</tr>
</tbody>
</table>
| 0.0r9    | 5/27/2021    | Bill Martin         | Incorporated Oscar Pinto/Bill Martin CSE/CSx proposal approved on 5/25
|          |              |                     | Moved all unresolved comments to editor’s notes
|          |              |                     | One exception is to add language about adding text on insufficient length into all appropriate APIs. This will be done prior to release. |
| 0.0r10   | 6/1/2021     | Bill Martin         | Added changes to how length is handled if too short
|          |              | Oscar Pinto         | Changes proposed by Oscar Pinto |
| 0.0r11   | 6/3/2021     | Bill Martin         | Removed change tracking
|          |              |                     | Updated references and added a few missing cross references |
| 0.5r0    | 6/9/2021     | Arnold Jones        | Formatted as a Working Draft for release outside the TWG per TWG approval ballot. |
| 0.5r1    | 06/22/2021   | Bill Martin         | Results of TWG meeting
|          |              |                     | Incorporated approved changes for Query properties |
| 0.5r2    | 12/14/2021   | Bill Martin         | Accepted previous changes
|          |              |                     | Incorporated approved changes for configuration |
| 0.5r3    | 02/11/2022   | Bill Martin         | Accepted previous changes
|          |              |                     | Incorporated Approved changes from Oscar Pinto |
| 0.5r4    | 02/18/2022   | Bill Martin         | Accepted previous changes
|          |              |                     | Incorporated Approved changes from Oscar Pinto
<p>|          |              |                     | Updated all occurrences to “CSxes” |
| 0.5r5    | 05/27/2022   | Bill Martin         | Accepted previous changes |</p>
<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Name</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5r6</td>
<td>06/22/2022</td>
<td>Bill Martin</td>
<td>Accepted all previous changes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Incorporated device discovery proposal</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Incorporated editor’s note review changes</td>
</tr>
<tr>
<td>0.8r0</td>
<td>06/29/2022</td>
<td>Bill Martin</td>
<td>Accepted all previous changes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Updated revision for Public Review and RFC</td>
</tr>
</tbody>
</table>

Suggestions for revisions should be directed to http://www.snia.org/feedback/.
Table of Contents

1 SCOPE................................................................................................................................. 14
  1.1 ABOUT COMPUTATIONAL STORAGE APIs ................................................................. 14
  1.2 DOCUMENT LAYOUT ................................................................................................. 14

2 DEFINITIONS, ABBREVIATIONS, AND CONVENTIONS.................................................. 15
  2.1 DEFINITIONS .................................................................................................................. 15
    2.1.1 Allocated Function Data Memory ............................................................................. 15
    2.1.2 Cluster ..................................................................................................................... 15
    2.1.3 Computational Storage ............................................................................................ 15
    2.1.4 Computational Storage Array ................................................................................... 15
    2.1.5 Computational Storage Device ............................................................................... 16
    2.1.6 Computational Storage Drive .................................................................................. 16
    2.1.7 Computational Storage Engine ............................................................................... 16
    2.1.8 See SNIA Computational Storage Architecture and Programming Model
          Computational Storage Function .................................................................................. 16
    2.1.9 Computational Storage Function Memory ............................................................... 16
    2.1.10 Computational Storage Processor .......................................................................... 17
    2.1.11 Computational Storage Resource (CSR) ................................................................. 17
    2.1.12 Container ................................................................................................................ 17
    2.1.13 CSE name ............................................................................................................... 17
    2.1.14 Filesystem .............................................................................................................. 17
    2.1.15 Function Data Memory ........................................................................................... 18
    2.1.16 host ......................................................................................................................... 18
    2.1.17 Hypervisor ............................................................................................................. 18
    2.1.18 Key Value .............................................................................................................. 18
<table>
<thead>
<tr>
<th>Section</th>
<th>Function Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.10.4</td>
<td>csHelperReconfigureBatchEntry()</td>
<td></td>
<td>98</td>
</tr>
<tr>
<td>6.10.5</td>
<td>csHelperResizeBatchRequest()</td>
<td></td>
<td>98</td>
</tr>
<tr>
<td>6.10.6</td>
<td>csQueueBatchRequest()</td>
<td></td>
<td>99</td>
</tr>
<tr>
<td>6.11</td>
<td>EVENT MANAGEMENT</td>
<td></td>
<td>103</td>
</tr>
<tr>
<td>6.11.1</td>
<td>csCreateEvent()</td>
<td></td>
<td>103</td>
</tr>
<tr>
<td>6.11.2</td>
<td>csDeleteEvent()</td>
<td></td>
<td>103</td>
</tr>
<tr>
<td>6.11.3</td>
<td>csPollEvent()</td>
<td></td>
<td>104</td>
</tr>
<tr>
<td>6.12</td>
<td>MANAGEMENT</td>
<td></td>
<td>106</td>
</tr>
<tr>
<td>6.12.1</td>
<td>csQueryDeviceProperties()</td>
<td></td>
<td>106</td>
</tr>
<tr>
<td>6.12.2</td>
<td>csQueryDeviceCapabilities()</td>
<td></td>
<td>107</td>
</tr>
<tr>
<td>6.12.3</td>
<td>csQueryDeviceStatistics()</td>
<td></td>
<td>108</td>
</tr>
<tr>
<td>6.12.4</td>
<td>csSetDeviceCapability()</td>
<td></td>
<td>109</td>
</tr>
<tr>
<td>6.12.5</td>
<td>csDownload()</td>
<td></td>
<td>110</td>
</tr>
<tr>
<td>6.12.6</td>
<td>csConfig()</td>
<td></td>
<td>111</td>
</tr>
<tr>
<td>6.12.7</td>
<td>csAbortCSE()</td>
<td></td>
<td>113</td>
</tr>
<tr>
<td>6.12.8</td>
<td>csResetCSE()</td>
<td></td>
<td>113</td>
</tr>
<tr>
<td>6.13</td>
<td>STREAM MANAGEMENT</td>
<td></td>
<td>114</td>
</tr>
<tr>
<td>6.13.1</td>
<td>csAllocStream()</td>
<td></td>
<td>114</td>
</tr>
<tr>
<td>6.13.2</td>
<td>csFreeStream()</td>
<td></td>
<td>115</td>
</tr>
<tr>
<td>6.14</td>
<td>LIBRARY MANAGEMENT</td>
<td></td>
<td>116</td>
</tr>
<tr>
<td>6.14.1</td>
<td>csQueryLibrarySupport()</td>
<td></td>
<td>116</td>
</tr>
<tr>
<td>6.14.2</td>
<td>csQueryPlugin()</td>
<td></td>
<td>117</td>
</tr>
<tr>
<td>6.14.3</td>
<td>csRegisterPlugin()</td>
<td></td>
<td>118</td>
</tr>
<tr>
<td>6.14.4</td>
<td>csDeregisterPlugin()</td>
<td></td>
<td>119</td>
</tr>
<tr>
<td>A</td>
<td>SAMPLE CODE</td>
<td></td>
<td>120</td>
</tr>
</tbody>
</table>
A.1  Initialization and queuing a synchronous request ........................................ 120
A.2  Queuing an asynchronous request .................................................................. 121
A.3  Using batch processing .................................................................................... 122
A.4  Applying hybrid batch processing feature ......................................................... 124
A.5  Using files for storage IO .................................................................................. 126
Table of Figures

Figure 1: An Architectural view of Computational Storage ................................................................. 22
Figure 2: API interrelationships ........................................................................................................... 27
Figure 3: Example API flows ............................................................................................................... 35
Figure 4: System Memory Map ............................................................................................................ 40
Figure 5: Example data transfers between AFDM in a CSx and host memory .................................... 41
Figure 6: Batch requests ....................................................................................................................... 43
Figure 7: Optimal CSF Scheduling ....................................................................................................... 48
Figure 8: CSF Grouping ......................................................................................................................... 49
Figure 9: API access flows .................................................................................................................... 52
Figure 10: Resource dependency chart ............................................................................................... 73
1 Scope

This document describes the software application interface definitions for a Computational Storage device CSx. This is the base set of functions and additional libraries are able to be built on this base set of functions.

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

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

Editor's note – The following need to be added to complete this draft:

1.1 About Computational Storage APIs

Computational Storage (CS) APIs are targeted towards providing a standardized way to access compute offload capable devices. This API specification is based on the SNIA Computational Storage Architecture and Programming Model. These may be connected direct attached or network attached or attached over some kind of fabric. This specification of CS APIs targets both types of connected devices with the aim of providing an interface that is seamless while standardized across all such current and future Computational Storage Devices.

Additionally, the CS APIs 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 substitutes seamlessly without changing the application. Additional interfaces are provided for this level of substitution.

This is a work in progress and there are expected to be enhancements and changes to the APIs defined in this specification.

1.2 Document layout

This document is broken down by providing a familiarity of device types, API usages, API 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 an API

Note 1 to entry:
See SNIA Computational Storage Architecture and Programming Model

2.1.2 Cluster
functional collection of servers or hosts

2.1.3 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 Programming Model

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

Note 1 to entry:
See SNIA Computational Storage Architecture and Programming Model
2.1.5 Computational Storage Device
Computational Storage Drive (CSD), Computational Storage Processor (CSP), or Computational Storage Array (CSA).

Note 1 to entry:
See SNIA Computational Storage Architecture and Programming Model

2.1.6 Computational Storage Drive
storage element that provides Computational Storage Functions and persistent data storage.

Note 1 to entry:
See SNIA Computational Storage Architecture and Programming Model

2.1.7 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:

2.1.8 See SNIA Computational Storage Architecture and Programming Model
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 1 to entry:
See SNIA Computational Storage Architecture and Programming Model

2.1.9 Computational Storage Function Memory
Device memory for storing Computational Storage Functions (CSFs)

Note 1 to entry:
See SNIA Computational Storage Architecture and Programming Model
2.1.10 **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 Programming Model

2.1.11 **Computational Storage Resource (CSR)**

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: CSE, CPU, memory, and FPGA resources

Note 3 to entry:

See SNIA Computational Storage Architecture and Programming Model

2.1.12 **Container**

A Docker container does not host a VM but instead binds an application to a (Docker) library that provides a secure container-type environment to the application and host OSs. It uses fewer resources and is lightweight compared to a conventional Hypervisor/VM configuration

2.1.13 **CSE name**

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

2.1.14 **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 abstract named data objects of variable size called files
2.1.15 **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

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

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

2.1.18 **Key Value**
storage that stores and retrieves user data based on a key that is associated with that data

2.1.19 **NVMe®**
NVM Express Specification

2.1.20 **Peer-to-Peer**
data transfer directly between two devices that does not involve a host or host memory

2.1.21 **P2P**
Peer-to-Peer

2.1.22 **PCIe®**
Peripheral Component Interconnect Express is a high-speed serial computer expansion bus standard

2.1.23 **Virtual Machine**
virtual machine or guest OS within a virtualized environment
2.2 Keywords

In the remainder of the specification, 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 *need not*
keywords indicating a feature that is not required to be implemented; “need not” is equivalent to “is not required to”

2.2.5 *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 it shall be implemented as defined in this standard

2.2.6 *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

2.2.7 *should*
a keyword indicating flexibility of choice with a strongly preferred alternative

2.3 Abbreviations

AFDM Allocated Function Data Memory
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 Architecture and Programming Model
3 Computational Storage

These APIs provide definitions of functions to support the SNIA Computational Storage Architecture specification.

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

CSxes as defined in the SNIA Computational Storage Architecture specification includes Computational Storage Processors (CSP), Computational Storage Drives (CSD) and Computational Storage Arrays (CSA) (see Figure 1)
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 table below provides examples of computational storage functions.

**Table 1: Example Computational Storage Function Types**

| Example of CSFs |
Compression

Encryption

Database filter

Erasure coding

RAID

Hash/CRC

RegEx (pattern matching)

Scatter Gather

Pipeline

Video compression

Data Deduplication

Large Data Set

Table 2: Example Computational Storage Engine Environment Types

Enumerated Image types

Operating System Image Loader

Container Image Loader

Berkeley packet filter (BPF) loader

FPGA Bitstream loader

Editors note: Need to determine how we query for a specific application classification and specific characteristics

Computational Storage defines 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.
This document defines the host level interfaces at the application level using software APIs.

Editor's Note: Add an ecosystem discussion here.
4 APIs Overview

Computational storage is possible with CSEs that are able to execute compute tasks typically run on a host CPU. These CSEs may use FDM that is different from the host memory and memory for storing CSFs. This memory is where computation programs run when they do. A mechanism is needed to transfer data to and from AFDM. 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 APIs that target these operations and interactions with the CSE. This section targets the usage of APIs 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. There are additional APIs required for a fully device managed interface. A future revision of this standard will also cover how the device may manage the FDM without host control.

![Figure 2: CS API Library](image)
Although the APIs have been tailored for a host managed interface, they also apply to a device managed interface. In the device managed interface, the APIs are performed by the device. Discovery, access, allocation and configuration of resources and all queued operations directly apply. Only the completion models may need host support to map them (e.g., callback vs. synchronous model).

As Computational Storage APIs provide mechanisms to allocate AFDM, there is a requirement that the case of computation overrunning the AFDM needs to be documented. This will be documented in a future revision of this standard.

If the device that this API interfaces to does not implement a particular function the API may return an error or implement an emulation of the function. The default is to return an error. The mechanism to control which of these options is performed is to be defined in a future revision of this standard.

The interrelation between applications, APIs, CSxes, and functions is shown in Figure 3.
Figure 3: API interrelationships
4.1 Discovery and configuration

As shown in Figure 3, 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 6.12.6). The CSx may be directly attached to the host or connected through a network or fabric. This document and APIs is storage agnostic. This API provides mechanisms for discovery of functionality of CSxes but not discovery of CSxes. We use NVMe as the exemplar throughout.

Figure 1Figure 4: Computational Storage Architecture
4.1.1 Discovery

CSxes may be discovered using the csQueryCSxList() API. The API returns a comma separated list of CSxes if there are more than one available. A CSx may also be discovered using the csGetCSxFromPath() API, where the path represents a device, directory or file.

Once a CSx is discovered, its resources may be queried with the csQueryDeviceProperties() API. This API provides individual properties available at different resource levels such as the CSx, CSE, CSEE and CSF. The API also provides details on the repository, activation states and configuration. Figure a1 illustrates how the API may be applied at various resource levels by providing the resource identifier as the input. The engine type here is an abstracted representation of the compute hardware type resource and is specific to a device as provided by the vendor. It is uniquely identified by the field CSETypeToken. A vendor may chose not to expose engine differences.

![Figure 5: CSx resource overview](image)

The API takes the resource type as input and provides the necessary property as output. The input of CS_CSx_TYPE provides CsxProperties data structure as output which denotes hardware and software details of the CSx. Similarly, other resource types may be provided as input to get the necessary outputs as shown in Table b1.
### Table 3: Device properties by resource type

<table>
<thead>
<tr>
<th>CSRESOURCE_TYPE</th>
<th>Input</th>
<th>Properties Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_CSx_TYPE</td>
<td>CSxProperties</td>
<td></td>
</tr>
<tr>
<td>CS_CSE_TYPE</td>
<td>CSEProperties</td>
<td></td>
</tr>
<tr>
<td>CS_CSEE_TYPE</td>
<td>CSEEProperties</td>
<td></td>
</tr>
<tr>
<td>CS_CSF_TYPE</td>
<td>CSFProperties</td>
<td></td>
</tr>
<tr>
<td>CS_VENDOR_SPECIFIC_TYPE</td>
<td>CSVendorSpecific</td>
<td></td>
</tr>
</tbody>
</table>

Additional details on the properties data structures and their sub-structures are provided in 6.3.4.2.1.

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

```c
// 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);
    // query for CSE properties
    status = csQueryDeviceProperties(devArray[i], CS_CSE_TYPE, &lenCSE, propCSE);
    // query for CSEE properties
    status = csQueryDeviceProperties(devArray[i], CS_CSEE_TYPE, &lenCSEE, propCSEE);
    // query for CSF properties
    status = csQueryDeviceProperties(devArray[i], CS_CSF_TYPE, &lenCSF, propCSF);
    // loop through the whole list
    token = strtok(NULL, "\,");
    i++;
}
```
4.1.2 Configuration

A CSx needs to be configured to be usable. Once it has been fully discovered, it may be configured using the `csConfig()` API. Configuration involves activation of the specific resource. The API takes the `CsConfigInfo` data structure as input to configure the specific resource.

Each resource is identified by its `Id` field in its 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 must be configured as activated before it may be used. A CSEE is activated by pairing its `CSEEId` in `CSEEInfo` data structure with a `CSEId` located in `CSEInfo` data structures. More than one engine type may be paired with a CSEE by activating the `Id` pairs. Figure a2 depicts a CSEE being paired with a CSE whose `Ids` are provided as input, and on successful activation return `EEId`, the activated CSEE instance.

![Figure 6: Activating a CSEE](image)

An `EEId` is used to configure a CSF. The CSEE activated instance informs the device that it will be utilized for computational storage activities. The device in turn will setup internal configuration options and resources to bring it to this state.

4.1.2.2 Configuring a CSF

Only activated CSFs may be used during execution. Activation of a CSF involves it being paired with an active CSEE instance and one or more compute resources.

As input, a previously activated CSEE instance `EEId` is paired with a CSF using its `CSFId` and one or more compute resource represented by `ERId`. Figure a3 depicts the flow with the various inputs and on successful activation provide `FIId` the activated CSF instance as output.
Activated resources will consume device resources as part of their internal configuration. CSF’s and CSEE’s may also be deactivated using `csConfig()` API. 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. Here, the built-in CSF may be deactivated to conserve device resources.

Additional details on configuration data structures are provided in <6.3.4.2.2 Configuration Data Structures>

### 4.2 Discovering CSFs

CSFs are not executable before activation. The steps described above to configure CSFs for activation must be followed to achieve this. Only activated CSFs may be discovered using the `csGetCSFId()` and `csQueryCSFList()` APIs. Figure a4 summarizes the APIs required to discover and configure a CSx and its resources. Only activated CSF instances denoted by their FIIId’s will be populated by the CSF APIs.
4.3 FDM allocation

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

FDM is able to be allocated and deallocated using the `csAllocMem()` and `csFreeMem()` APIs.

4.4 Compute types and execution

CSEs that are able to perform compute offload may be of various types (e.g., ASICs, FPGAs, and embedded CPUs). `csQueueComputeRequest()` and `csQueueBatchRequest()` request execution of compute operations independent of the type of CSEs. The type specific functionality of a CSE may be handled by a device driver whose implementation details may be abstracted at the API 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 as the application does not have to change. Additional details on plugins is available in section 6.14.

4.5 Downloading Functions

In certain CSEs, compute programs are able to be downloaded. `csDownload()` provides the mechanism for CSEs with such capabilities. A compute program downloaded may
contain one or more computational storage functions. Following a download, the host may initiate a discovery to determine what CSFs are available.

4.6 Extending API support

The plugin capability of this specification provides the ability to extend capabilities.

Plugins may be applied at various places in a CS API software stack implementation to provide features and help support a common set of APIs. Plugins are required to be registered first with the API library before they are able to be applied. The APIs `csQueryPlugin()` and `csQueryLibrarySupport()` provide details on plugins instantiated in the library and APIs `csRegisterPlugin()` and `csDeregisterPlugin()` are used to insert/remove plugin capability in the CS API stack.

4.7 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, P2P has the possibility of failing since data may not get loaded or stored in the right 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, IO 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.

4.8 Extending API support

A plugin is a software entity that provides the data exchange between the abstracted CS APIs and a device's specific interface. They do so by having a mapping layer between these interfaces. A Plugin may also abstract specific functionality for a device. Plugin’s also play a role in providing seamless access, e.g., local or remote connectivity using the same APIs, supporting new features, substituting/aiding in device feature support.

4.9 API usage example

The following example illustrates the usage of CS APIs 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 9 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.
6) The output results are copied from the output AFDM to host memory.

Figure 9: Example API 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 APIs \texttt{csAllocMem()} and \texttt{csFreeMem()} 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 API 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 APIs.

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 \texttt{csAllocMem()} API.

The loading of storage media into the allocated AFDM is conducted by \texttt{csQueueStorageRequest()} API.

Compute functions provided with these buffers are executed using the \texttt{csQueueComputeRequest()} and \texttt{csQueueBatchRequest()} APIs.

Data transfers between AFDM and host memory are conducted using the \texttt{csQueueCopyMemRequest()} API.

Key resources utilized by CSFs are 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 that the CSF will work on
   B) Input parameters to the CSF
b) Data transfer from FDM to host memory
   A) Data that the CSF returns to host application
   B) Miscellaneous results (e.g., status and other variables)
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 local RAM for device based functions etc)

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. In typical architectures that are not enabled with computational storage, the host reads data from the SSD into host memory and then copies it over to FDM on the CSD. 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 needs to be written to the SSD, it first gets DMAed to host memory and then written to SSD. Each of these operations require 2 data transactions on the fabric and in doing so consume a part or whole 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 csAllocMem() 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 9 depicts a CSD that does not expose FDM in the host’s address space. csAllocMem() allocates FDM at a granularity as specified by the CSx. This API in addition to allocating FDM also facilitates mapping it into host’s system address space, if the CSx supports this mapping.

5.1.2.1 When to map AFDM to a virtual address

AFDM when allocated should be requested for host address mapping only in the following conditions:

a) AFDM will be passed to the OS filesystem/block subsystem to load data directly from the SSD utilizing the P2P protocol
b) AFDM will be passed to the OS filesystem/block subsystem to commit data directly from CSx to SSD using P2P; and
c) AFDM will be accessed directly from host application software

The allocation request for mapping however depends on the CSx to have that capability of exposed FDM into 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, csQueueCopyMemRequest() provides a mechanism for data transfer. Direct device memory access when mapped may result in unpredictable results in certain configurations (e.g. with hypervisors DMA may result in errors). In these cases device memory should be accessed through the device DMA engine using this API.

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 csQueryDeviceProperties() to verify which modes it supports.

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 csAllocMem() do 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 API 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 csQueueStorageRequest() which facilitates the transfer of data from storage directly to FDM buffers where the transfers do not leave the device. Doing so may save on host’s CPU, cache and memory usage and the fabric bandwidth which translate into performance, latency and power benefits.

5.1.3.2 **FDM exposed to host address space**

The API definitions support devices that also expose FDM to host address space. Here a virtual address pointer is requested during allocation through 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 in some cases external fabric transactions.

csAllocMem() 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 10, devices operate with host CPU by exposing AFDM in host’s address space (e.g., the NVMe and CSx both make their memory visible through PCIe BARs). 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 csAllocMem() 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 it 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;

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 2 needs to be passed directly through an ioctl call to the NVMe driver. Here the SSD block translations may need to be done from the appropriate filesystem to describe the I/O request at the block level;
Figure 10: 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 it is in AFDM; and
f) When compute is complete, the CSF passes the data back to the application memory either through `csQueueCopyMemRequest()` or committing it 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()` API call.

This API takes data transfer direction as part of the request as shown in Figure 11.
Figure 11: Example data transfers between AFDM in a CSx and host memory

5.2 Scheduling Compute Offload Jobs

Scheduling compute offload is done with `csQueueComputeRequest()`. This request takes as input the CSF to queue the job to with its arguments. The number of arguments and their values should match the definition of the offload function as the runtime will not enforce this and the behavior is undefined.

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

Compute offload jobs require input and produce output. Each of these entities require a job request.

```
input  compute  output
```

The Table 4 summarizes job processing for input, compute and output.
### Table 4: Job request processing

<table>
<thead>
<tr>
<th>Job</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong></td>
<td>Provides input to a compute job. Input to a compute job may be provided in two ways:</td>
</tr>
<tr>
<td></td>
<td><strong>Input method</strong></td>
</tr>
<tr>
<td><strong>Storage</strong></td>
<td>a) Use file system calls with device memory mapped to host; and</td>
</tr>
<tr>
<td></td>
<td>b) Use csQueueStorageRequest() with type option CS_STORAGE_LOAD_TYPE</td>
</tr>
<tr>
<td><strong>Host memory</strong></td>
<td>Use csQueueCopyMemRequest() or csQueueBatchRequest() with option CS_COPY_TODEVICE</td>
</tr>
<tr>
<td><strong>Compute</strong></td>
<td>The actual compute job may be scheduled to run in the following ways:</td>
</tr>
<tr>
<td></td>
<td><strong>method</strong></td>
</tr>
<tr>
<td><strong>Single or batch request</strong></td>
<td>Use csQueueComputeRequest() for a single request or csQueueBatchRequest() for batch request.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td>Provides output from a compute job. Output from a compute job may be received in two ways:</td>
</tr>
<tr>
<td></td>
<td><strong>Input method</strong></td>
</tr>
<tr>
<td><strong>Storage</strong></td>
<td>a) Use file system calls with device memory mapped to host; and</td>
</tr>
<tr>
<td></td>
<td>b) Use csQueueStorageRequest() with type option CS_STORAGE_STORE_TYPE</td>
</tr>
<tr>
<td><strong>Host memory</strong></td>
<td>csQueueCopyMemRequest() or csQueueBatchRequest() with option CS_COPY_FROM_DEVICE</td>
</tr>
</tbody>
</table>
5.2.1 Batch requests

The API `csQueueBatchRequest()` is an advanced queuing mechanism that minimizes the interactions between host software and device by optimizing the input(s) and output(s). It is useful in cases where the work required to be performed by the CSx must be done in a particular order with a set of operations. These could be serialized jobs, parallelized jobs or a combination of jobs 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 batched requests are done.

Batching requests using this API helps the application in pipelining multiple requests by their dependencies, reduce host cpu usage, reduce latencies by having less host context switches and overall having 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 it is not supported in hardware, the underlying software implementation of the APIs supports this usage and APIs and provide similar functionality. Batch request functionality is able to be discovered with `csQueryDeviceProperties()`.

![Figure 12: Batch requests](image)

Figure 12 illustrates different types of batch requests. In option a, a serialized notation of job requests using the batching option is shown. Here, an input data on completion feeds into a compute job whose output is fed into an output job. 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 Table 4.

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 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.

![Diagram of serialized operations example](image)

Applying it with function calls, there are many combinations of these jobs. A serial batch request presents jobs as an array with the order required. Serial batch request implies dependency between the previous and next job and does not require additional dependency details as a hybrid operation does (see 5.2.1.3).

In this example, 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.

![Serial batch request example](image)

The next example is the same as the previous with the addition of copying the results back to host memory. This example demonstrates an input job, a compute job and an output job.

![Example with copying results back to host](image)

The following example is a typical flow that manipulates stored data and provides the output back to host.

![Typical flow example](image)

In the following example, the output of a compute request becomes the input to the next compute request.
Parallelized operations examples

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

In this example, 6 compute jobs 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.

In another example usage, 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.

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 API 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 it 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 example shown below shows such a case. This is also depicted in Figure 12 option e.

Since a data dependency exists, and the data resides in device space, it is able to be provided as an input to enable hybrid mode with \texttt{csQueueBatchRequest()}. 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 function \texttt{csQueueBatchRequest()} itself 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 it 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.

In this example, each request represents a node in the batch of requests with \texttt{Mode} specifying the operation for that node.

Table 5 summarizes batch operations.
Table 5: Batch requests

<table>
<thead>
<tr>
<th>Batch mode</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>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 APIs that are able to be batched serially are csQueueStorageRequest(), csQueueComputeRequest() and csQueueCopyMemRequest().</td>
</tr>
<tr>
<td>parallel</td>
<td>In this mode of execution, the intended purpose is to breakdown 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 APIs csQueueStorageRequest(), csQueueComputeRequest() or csQueueCopyMemRequest() is able to be batched in a single request to execute in parallel. These APIs may also be mixed together and also run in parallel. The supporting hardware must support the required parallelism for this batch operation to execute as intended.</td>
</tr>
<tr>
<td>hybrid</td>
<td>In this mode, complex and nested operations are able to be performed with the batch request. The APIs csQueueStorageRequest(), csQueueComputeRequest() and csQueueCopyMemRequest() are able to be batched in a single request to execute in batch mode. The sequence of requests may be included as single requests or as a series of nested graph operations.</td>
</tr>
</tbody>
</table>

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

5.2.2 Optimal Scheduling

Batch based scheduling requests provide optimal IO 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 it.
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 must be used to provide hints during scheduling. Grouping is achieved by the function `csGroupComputeByIds()`.

When compute has to be aligned to utilize a CSF without idle time in-between executions, the scheduler by design should manage the transitions between different execution times most efficiently.

![Diagram](image)

**Figure 13: Optimal CSF Scheduling**

Figure 13 depicts compute being utilized very 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 Grouping CSFs

CSF functionality depends on its CSE implementation. Since CSEs can be different from one another, a CSF for one type of CSE may not have similar characteristics to a CSF for another type of CSE. CSFs for some CSEs like an embedded CPU may be able to use one image instance to invoke more than one functional instance to provide parallelism. But that may not be the case with CSFs that require physical image instances to represent more than one functional instance for parallelism (e.g., FPGAs and hardware ASICs). These CSFs may be represented instead as multiple physical instances when downloaded. In some cases, depending on the tools used to build these CSFs for these CSEs, there may be multiple CSFs that have the same functionality, but with different names. This may primarily arise as a build limitation on the maximum parallel CSFs that are able to fit in a downloadable program for this CSE. In this situation, the user has to be made aware of these subtle differences in CSFs and their association by names. Scheduling compute jobs to these CSFs requires the application to understand the limitation of maximum parallelism possible and requires managing these additional CSFs that have the same functionality individually.

Grouping of CSFs provides the user the ability to associate CSFs with similar functionality together. This involves querying the full list of CSFs using `csQueryDeviceForComputeList()` and then grouping them together using `csGroupComputeByIds()`.

Grouping compute functions provides a hint when scheduling compute jobs to any parallel CSFs within the group. A job when queued to a particular function by name is able to be scheduled to any parallel CSF within the group that is idle. All functions within the group must contain the same functionality.

Figure 14: CSF Grouping

Figure 14 illustrates a grouping policy applied, where function1 that has 4 parallel CSFs is grouped together with function2 that also has 4 parallel CSFs.

From the application’s perspective, using either function provides the full group’s capability in scheduling compute jobs. If the application chose function1 and schedules a 5th compute offload job when 4 are already outstanding, the scheduler uses the group hint and uses the capabilities of function2 to schedule the next job. Alternatively, the scheduler is able to optimize which job to schedule next to each available function as they become free.

The call to `csGroupComputeByIds()` is a privileged function restricted to applications with administrative rights. This API should be executed when the system is setup and before any compute jobs are run and may be part of the system policy.
5.4 Completion Models

Storage, Memory Copy and Compute requests use a queued IO 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 before it is completed from 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 request completes. With events, the user may poll using `csPollEvent()` API in the callers context and when ready to process the completion. The IO for both asynchronous completion types get a completion back only when the request at the device is complete.

Table 6: Completion Models

<table>
<thead>
<tr>
<th>Completion Model</th>
<th>Inputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
<td>Context = NULL</td>
<td>This is a blocking model, where the submitted request will not return to caller until complete.</td>
</tr>
<tr>
<td></td>
<td>CallbackFn = NULL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EventHandle = NULL</td>
<td></td>
</tr>
<tr>
<td>Asynchronous Callback</td>
<td>Context = &lt;User Context&gt;</td>
<td>This is a non-blocking model, where the user callback function is notified when the requested IO is complete.</td>
</tr>
<tr>
<td></td>
<td>CallbackFn = &lt;User Callback Function&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EventHandle = NULL</td>
<td></td>
</tr>
<tr>
<td>Asynchronous Event</td>
<td>Context = &lt;User Context&gt;</td>
<td>This is a non-blocking model, where the user event is signaled when the requested IO is complete. The user is able to poll the event handle for completion status to change from <code>CS_QUEUED</code>.</td>
</tr>
<tr>
<td></td>
<td>CallbackFn = NULL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EventHandle = &lt;User Event handle&gt;</td>
<td></td>
</tr>
</tbody>
</table>
6 CS API Interface Definitions

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

6.1 API Access and flow conventions

The API definitions listed in this section use the following convention.

Handles have very specific usage. Only one handle is accepted per task as the main input and additional handles will be referenced either as arguments or internally based on reference.
6.2 Usage Overview

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

Editor’s Note: Needs updates when API updates are complete

Table 7: CS API matrix

<table>
<thead>
<tr>
<th>Functionality</th>
<th>APIs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Device Discovery</td>
<td>csQueryCSxList()</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>- Identify CSxes</td>
<td></td>
</tr>
<tr>
<td>- Identify CSFs</td>
<td>csQueryCSFList()</td>
</tr>
<tr>
<td>- Identify CSx associated with Storage device</td>
<td>csGetCSxFromPath()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Device Access</th>
<th>csOpenCSx()</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Open/Close CSx device for access</td>
<td>csCloseCSx()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FDM management</th>
<th>csAllocMem()</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Allocate/Deallocate FDM</td>
<td>csFreeMem()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Storage IOs</th>
<th>Use filesystem with FDM and initiate P2P</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Issue read/write IOs from/to Storage</td>
<td>csQueueStorageRequest()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CSx Data movement</th>
<th>csQueueCopyMemRequest()</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Transfer data between device memory and host memory</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CSF access and scheduling</th>
<th>csGetCSFId()</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Schedule CSF on device</td>
<td></td>
</tr>
</tbody>
</table>

|                                               |              |
|                                               | csAbortCSF() |
|                                               | csQueueComputeRequest() |
|                                               | csHelperSetComputeArg() |
|                                               | csQueueBatchRequest() |
|                                               | csAllocBatchRequest() |
|                                               | csFreeBatchRequest() |
|                                               | csAddBatchEntry() |
|                                               | csHelperReconfigureBatchEntry() |
|                                               | csHelperResizeBatchRequest() |
## Device Management
- Query device properties and capabilities
- Manage device functionality

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>csQueryDeviceProperties()</td>
</tr>
<tr>
<td>csQueryDeviceCapabilities()</td>
</tr>
<tr>
<td>csQueryDeviceStatistics()</td>
</tr>
<tr>
<td>csGroupComputeByIds()</td>
</tr>
<tr>
<td>csUngroupComputeFromGroupId()</td>
</tr>
<tr>
<td>csSetDeviceCapability()</td>
</tr>
<tr>
<td>csDownload()</td>
</tr>
<tr>
<td>csConfig()</td>
</tr>
<tr>
<td>csAbortCSE()</td>
</tr>
<tr>
<td>csResetCSE()</td>
</tr>
<tr>
<td>csRegisterNotify()</td>
</tr>
<tr>
<td>csDeregisterNotify()</td>
</tr>
</tbody>
</table>

## Event Management
- Create/delete events for completion processing

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>csCreateEvent()</td>
</tr>
<tr>
<td>csDeleteEvent()</td>
</tr>
<tr>
<td>csPollEvent()</td>
</tr>
</tbody>
</table>

## Stream Management
- Allocate/free streams for stream based processing

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>csAllocStream()</td>
</tr>
<tr>
<td>csFreeStream()</td>
</tr>
</tbody>
</table>

## Library Management
- Query API library support
- Manage library interfaces to support APIs

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>csQueryLibrarySupport()</td>
</tr>
<tr>
<td>csQueryPlugin()</td>
</tr>
<tr>
<td>csRegisterPlugin()</td>
</tr>
<tr>
<td>csDeregisterPlugin()</td>
</tr>
</tbody>
</table>
6.3 Common Definitions

6.3.1 Data Types

6.3.1.1 Scalar data types

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s32</td>
<td>Signed 32-bit data; used as input to functions and arguments</td>
</tr>
<tr>
<td>u32</td>
<td>Unsigned 32-bit data; used in arguments scheduling a CSF</td>
</tr>
<tr>
<td>f32</td>
<td>Float 32-bit data; used in arguments scheduling a CSF</td>
</tr>
<tr>
<td>s64</td>
<td>Signed 64-bit data; used as input to functions and arguments</td>
</tr>
<tr>
<td>u64</td>
<td>Unsigned 64-bit data; used in arguments scheduling a CSF</td>
</tr>
<tr>
<td>f64</td>
<td>Float 64-bit data; used in arguments scheduling a CSF</td>
</tr>
<tr>
<td>u128</td>
<td>Unsigned 128-bit data; used in arguments scheduling a CSF</td>
</tr>
</tbody>
</table>

6.3.1.2 Vector data types

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_MEM_HANDLE</td>
<td>A memory handle is used in arguments for scheduling a CSF and AFDM copies</td>
</tr>
<tr>
<td>CS_STREAM_HANDLE</td>
<td>A stream handle used in arguments for scheduling a CSF</td>
</tr>
</tbody>
</table>

6.3.2 Status Values

One or more of the values in Table 8 are returned by the interface APIs and are classified under CS_STATUS.
<table>
<thead>
<tr>
<th>Status Value Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_SUCCESS</td>
<td>The action was completed with success</td>
</tr>
<tr>
<td>CS_COULD_NOT_MAP_MEMORY</td>
<td>The requested memory allocated could not be mapped</td>
</tr>
<tr>
<td>CS_DEVICE_ERROR</td>
<td>The device is in error and is not able to make progress</td>
</tr>
<tr>
<td>CS_DEVICE_NOT_AVAILABLE</td>
<td>The CSx is unavailable</td>
</tr>
<tr>
<td>CS_DEVICE_NOT_READY</td>
<td>The device is not ready for any transactions</td>
</tr>
<tr>
<td>CSDEVICE_NOT_PRESENT</td>
<td>The requested device is not present</td>
</tr>
<tr>
<td>CS_ENODEV</td>
<td>The device name specified does not exist</td>
</tr>
<tr>
<td>CS_ENOENT</td>
<td>No such device, file or directory exists</td>
</tr>
<tr>
<td>CS_ENTITY_NOT_ON_DEVICE</td>
<td>The entity does not exist on requested device</td>
</tr>
<tr>
<td>CS_ENXIO</td>
<td>No Storage or CSE was available</td>
</tr>
<tr>
<td>CS_ERROR_IN_EXECUTION</td>
<td>There was an error that occurred in the execution path</td>
</tr>
<tr>
<td>CS_FATAL_ERROR</td>
<td>There was a fatal error that occurred</td>
</tr>
<tr>
<td>CS_HANDLE_IN_USE</td>
<td>The requested handle is already in use</td>
</tr>
<tr>
<td>CS_INVALID_HANDLE</td>
<td>An invalid handle was passed</td>
</tr>
<tr>
<td>CS_INVALID_ARG</td>
<td>One or more invalid arguments were provided</td>
</tr>
<tr>
<td>CS_INVALID_EVENT</td>
<td>The event specified was invalid</td>
</tr>
</tbody>
</table>
### CS_INVALID_ID
The specified input ID was invalid and does not exist

### CS_INVALID_LENGTH
The specified buffer is not of sufficient length

### CS_INVALID_OPTION
An invalid option was specified

### CS_INVALID_CSF_ID
The CSF identifier specified was invalid

### CS_INVALID_CSF_NAME
The CSF name specified does not exist or is invalid

### CS_IO_TIMEOUT
An IO submitted has timed out

### CS_LOAD_ERROR
The specified download could not be initialized

### CS_MEMORY_IN_USE
The requested memory is still in use

### CS_NO_PERMISSIONS
There were insufficient permissions to proceed with request

### CS_NOT_DONE
The request is not done

### CS_NOT_ENOUGH_MEMORY
There is not enough memory to satisfy the request

### CS_NO_SUCH_ENTITY_EXISTS
There is no such entry that exists

### CS_OUT_OF_RESOURCES
The system is out of resources to satisfy the request

### CS QUEUED
The request was successfully queued

### CS_UNKNOWN_MEMORY
The memory referenced was unknown

### CS_UNSUPPORTED
The request is not supported

#### 6.3.3 Notification Options
The following definitions specify the fixed defined values that can be specified as notification options as an input to `csRegisterNotify()`. The same values will be provided to the notification callback if invoked.
Table 9: Notification Value Definitions

<table>
<thead>
<tr>
<th>Status Value Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_NOTIFY_SYSTEM_ERROR</td>
<td>A system error has occurred</td>
</tr>
<tr>
<td>CS_NOTIFY_CSE_UNRESPONSIVE</td>
<td>The specified CSE is not responding normally and may be unusable</td>
</tr>
<tr>
<td>CS_NOTIFY_CSEE_UNRESPONSIVE</td>
<td>The specified CSEE is not responding normally and may be unusable</td>
</tr>
<tr>
<td>CS_NOTIFY_CSF_UNRESPONSIVE</td>
<td>The specified CSF is not responding normally and may be unusable</td>
</tr>
<tr>
<td>CS_NOTIFY_CSx_ADDED</td>
<td>A new CSx is available</td>
</tr>
<tr>
<td>CS_NOTIFY_CSx_REMOVED</td>
<td>A CSx is not available</td>
</tr>
<tr>
<td>CS_NOTIFY_CSF_ADDED</td>
<td>A new CSF was loaded</td>
</tr>
<tr>
<td>CS_NOTIFY_CSF_REMOVED</td>
<td>A CSF was unloaded</td>
</tr>
<tr>
<td>CS_NOTIFY_RESOURCE_WARNING</td>
<td>The CSx is running out of resources</td>
</tr>
<tr>
<td>CS_NOTIFY_DOWNLOAD_INFO</td>
<td>Additional information available for downloaded program</td>
</tr>
<tr>
<td>CS_NOTIFY_CONFIG_INFO</td>
<td>Additional information is available for downloaded configuration</td>
</tr>
</tbody>
</table>

6.3.4 Data Structures

6.3.4.1 Enumerations

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

6.3.4.1.1 CS_RESOURCE_TYPE

typedef enum {
    CS_CSx_TYPE = 1,
    CS_CSE_TYPE = 2,
    CS_CSEE_TYPE = 3,
};
CS-CSF_TYPE = 4,
CS_VENDOR_SPECIFIC_TYPE = 5
} CS_RESOURCE_TYPE;

6.3.4.1.2 CS_STATE

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

6.3.4.1.3 CS_DOWNLOAD_TYPE

typedef enum {
    CS_FPGA_BITSTREAM = 1,
    CS_BPF_PROGRAM = 2,
    CS_CONTAINER_IMAGE = 3,
    CS_OPERATING_SYSTEM_IMAGE = 4,
    CS_LARGE_DATA_SET = 5,
    CS_VENDOR_SPECIFIC_CSF = 6,
    CS_VENDOR_SPECIFIC_CSEE = 7
} CS_DOWNLOAD_TYPE;

Editor’s Note: Work on download and config to reflect: CSF/CSEE/VS;
    CSF: FPGA_bitstream; eBPF; Container; VS;
    CSEE: OS; Container Platform; eBPF environment; VS
Subtype – source; object code: executable;
Do we do activate, deactivate, delete in config or download

6.3.4.1.4 CS_CONFIG_TYPE

typedef enum {
    CS_CSEE_ACTIVATE = 1,
    CS-CSF_ACTIVATE = 2,
    CS_VENDOR_SPECIFIC = 3
} CS_CONFIG_TYPE;

6.3.4.1.5 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;

6.3.4.1.6 CS_STORAGE_REQ_Mode

typedef enum {
    CS_STORAGE_BLOCK_IO = 1,
    CS_STORAGE_FILE_IO = 2
} CS_STORAGE_REQ_Mode;
6.3.4.1.7 **CS_STORAGE_IO_TYPE**

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

6.3.4.1.8 **CS_COMPUTE_ARG_TYPE**

This enum defines the CSF argument types.

typedef enum {
    CS_AFDM_TYPE = 1,
    CS_32BIT_VALUE_TYPE = 2,
    CS_64BIT_VALUE_TYPE = 3,
    CS_STREAM_TYPE = 4,
    CS_DESCRIPTOR_TYPE = 5
} CS_COMPUTE_ARG_TYPE;

6.3.4.1.9 **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;

6.3.4.1.10 **CS_BATCH_REQ_TYPE**

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

6.3.4.1.11 **CS_STAT_TYPE**

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

Editor's note: These need further work (e.g., which CSE Stats are required from devices?)

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_FUNCTION = 3 // query statistics on a specific function
} CS_STAT_TYPE;
6.3.4.1.12  **CS_CAP_TYPE**

This data type defines the various capabilities that are able to be queried.

   Editor’s note: These need further work (these do not seem specific to CS)

```c
typedef enum {
    CS_CAPABILITY_CSx_TEMP = 1,
    CS_CAPABILITY_CSx_MAX_IOS = 2
    // TODO: define additional configuration options
} CS_CAP_TYPE;
```

6.3.4.1.13  **CS_STREAM_TYPE**

The data type **CS_STREAM_TYPE** is defined as follows:

```c
typedef enum {
    CS_STREAM_COMPUTE_TYPE = 1
} CS_STREAM_TYPE;
```

6.3.4.1.14  **CS_LIBRARY_SUPPORT**

   Editor’s note: These need further work – currently not used

```c
typedef enum {
    CS_FILE_SYSTEMS_SUPPORTED = 1,
    // TODO:
    CS_RESERVED = 2
} CS_LIBRARY_SUPPORT;
```

6.3.4.1.15  **CS_PLUGIN_TYPE**

```c
typedef enum {
    CS_PLUGIN_COMPUTE = 1,
    CS_PLUGIN_NVME = 2,
    CS_PLUGIN_FILE_SYSTEM = 4,
    CS_PLUGIN_CUSTOM = 8
    // TODO:
} CS_PLUGIN_TYPE
```

   Editor’s note: These need further work – currently not used

6.3.4.2  **Structures**

The structures in this section are used in API parameters and data structures.
6.3.4.2.1 Properties Data Structures

The data structure CSProperties is queried using csQueryDeviceProperties() and provides the properties for all compute resources of a CSx. The structure contains sub-structures that must be queried individually using the CS_Resource_Type enumerator. The sub-structures include CSxProperties, CSEProperties, CSEEProperties and CSFProperties.

The sub-structure CSxProperties provides information pertaining to the CSx and are fixed attributes that do not change while it is functional. The FDM.IsDeviceManaged field when set to 1 identifies that the CSx manages FDM for allocations and deallocations and shall decide how the memory is managed. If this field is set to zero, it means that the host shall manage this resource. The FDM.IsHostVisible 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 needs specific APIs to operate on it.

The sub-structure CSEProperties provides information on all CSEs, where each one is described by the sub-structure CSEInfo. The field CSE.TypeToken is a device specified entry that uniquely distinguishes between different CSE types. The NumBuiltInFunctions field denotes the total number of preloaded functions that were available with this CSE on reset. The MaxRequestsPerBatch field denotes the maximum number of requests that can be batched together in a batch request through csQueueBatchRequest API. 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 API. Each CSE is further identified by the sub-structure ComputeResource that provide individual details on the CSE.

The CSEEProperties sub-structure provides details on the execution environment and is associated to the CSE by the CSE.TypeToken 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 sub-structure CSFProperties describes all the CSFs that are available on the CSx. Each CSF is described by the sub-structure CSFInfo and include fields to uniquely identify the CSF and verify if it is built-in. A CSF must 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 that engine type allows it.

6.3.4.2.1.1 CSxProperties

typedef struct {
    u16 HwVersion;       // specifies the hardware version of this CSx
    u16 SwVersion;       // specifies the software version that runs on this CSx
    u16 VendorId;        // specifies the vendor id of this CSx
    u16 DeviceId;        // specifies the device id of this CSx
    char FriendlyName[32];       // an identifiable string for this CSx
    u32 CFMinMB;          // Amount of CFM in megabytes installed in device
    u32 FDMInMB;          // amount of FDM in megabytes installed in device
} CSxProperties;
struct {
    u64 FDMIsDeviceManaged : 1; // FDM allocations managed by device
    u64 FDMIsHostVisible : 1; // FDM may be mapped to host address space
    u64 BatchRequestsSupported : 1; // CSx supports batch requests in hardware
    u64 StreamsSupported : 1; // CSx supports streams in hardware
    u64 Reserved : 60;
} Flags;
} CSxProperties;

6.3.4.2.1.2 ComputeResource

typedef struct {
    u16 HwVersion;
    u16 SwVersion;
    char Name[32]; // an identifiable string for this CR
    u8 RelativePerformance; // values [1-10]; higher is better; 0 is not defined
    u8 RelativePower; // values [1-10]; lower is better; 0 is not defined
    u16 ERId; // Engine Resource Identifier for this ComputeResource
    CRProperties *Features; // additional features like perf, security etc [TBD]
} ComputeResource;

6.3.4.2.1.3 CSEInfo

typedef struct {
    u16 CSETypeToken; // device provided token to differentiate between its
    // CSE types
    u32 MaxRequestsPerBatch; // maximum number of requests supported per batch request
    u32 MaxCSFParametersAllowed; // maximum number of parameters supported
    u16 CSEId; // CSE Id 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;

6.3.4.2.1.4 CSEProperties

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

6.3.4.2.1.5 CSEEInstance

typedef struct {
    enum CS_STATE State; // current activation state
    u16 CSEId; // CSE Id unique to this CSx
    u16 EEIID; // Execution Environment Instance Identifier
} CSEEInstance;

6.3.4.2.1.6 CSEEInfo

typedef struct {
    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
U16 CSEEID; // unique CSEE Id
U16 MaxCSFs; // maximum number of CSFs for this CSEE
U16 NumActivatedCSFs; // number of activated CSFs
U16 NumEES; // number of activated CSEE instances
CSEEInstance *EEInstances; // a pointer to a list of activated CSEE instances
} CSEEInfo;

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

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

6.3.4.2.1.9 CSFInfo
typedef struct {
  u8 GlobalId[16]; // placeholder for global identifier
  char UniqueName[32]; // an identifiable string for this CSF, if available
  u16 CSETypeToken; // device provided token to differentiate between its
                     // CSE types
  u16 CSFId; // unique CSF Id
  u8 Builtin: 1; // preloaded by vendor
  u8 Reserved: 7;
  u16 NumFIs; // number of associated instances for this CSF
               // pointer to list of CSF instances with CSEE & CR details
  CSFInstance *FInstances;
} CSFInfo;

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

6.3.4.2.1.11 CsProperties
typedef union {
  CSxProperties CSxDetails; // details on CSx
  CSEEProperties CSEEDETAILS[1]; // details on all CSEs
  CSEEProperties CSEEDETAILS[1]; // details on all CSEEs
  CSFProperties CSFDetails[1]; // details on all CSFs
  CSVendorSpecific *VSDetails; // vendor specific
} CsProperties;
6.3.4.2.2 Configuration Data Structures

The data structure `csConfigInfo` is provided as input to configure a CSx using `csConfig()`. On success the data structure `CsConfigData` provides the results of the requested configuration. Configurations must be selected using `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 `CSEEActivateConfig` 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 is primarily used for activating a CSF.

Similarly, the CSF resource may be configured with a valid EEIId and one or more compute resource (CR). 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 with `csGetCSFId()` and `csQueryCSFList()`. Only activated CSF instances are used in execution using `csQueueComputeRequest()` or `csQueueBatchRequest()`.

6.3.4.2.2.1 CSEEActivateConfig

typedef struct {
    enum CS_STATE State;   // requested activation state
    u16 CSEEId;           // unique CSEE Identifier
    u16 CSEId;            // CSE Id unique to CSx
} CSEEActivateConfig;

6.3.4.2.2.2 CSFActivateConfig

typedef struct {
    enum CS_STATE State;   // requested activation state
    u16 CSFId;             // unique CSF Id
    u16 EEIId;             // Execution Environment Instance Identifier
    U16 NumCRs;            // number of CRs in array
    u16 CRArray[1];       // an array of one or more Compute Resources (CRs)
} CSFActivateConfig;

6.3.4.2.2.3 CsActivationInfo

typedef struct {
    enum CS_STATE State;   // current activation state
    u16 Id;                // resource specific unique Identifier
} CsActivationInfo;
6.3.4.2.2.4 CsConfigInfo

The data structure CsConfigInfo is defined as follows:

typedef struct {
    enum CS_CONFIG_TYPE Type;
    union {
        CSEEActivateConfig CSEEActivateInfo;       // configuration details for CSEE
        CSFActivateConfig CSFActivateInfo;         // configuration details for CSF
        CSVendorConfig *VSInfo;                   // vendor specific
    } u;
} CsConfigInfo;

6.3.4.2.2.5 CsConfigData

typedef struct {
    CsActivationInfo Data;
    void *VSData;
} CsConfigData;

6.3.4.2.3 Memory Data Structures

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

6.3.4.2.3.1 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;

6.3.4.2.3.2 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 {
    enum CS_MEM_COPY_TYPE Type;          // see 6.3.4.1.5
    union {
        void *HostVAddress;                // defines host memory if specified in Type
        CsDevAFDM SrcDevMem;               // defines the source device memory for copy
    } u;
    CsDevAFDM DevMem;                    // see 6.3.4.2.3.1
    unsigned int Bytes;
}
The structure `CsStorageRequest` describes the storage IO request between the storage device and the CSF. Storage IO is able to be described as a block or file request and utilizes the `Mode` field to select it. The `Type` field describes the direction of data flow from storage device.

For file requests, the `CsFileIo` structure describes the file request to perform with details on file handle, offset within 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 must 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 must adhere to the storage drives block requirements. For file write based requests, the API will synchronize on writing to that portion of the file with the filesystem and reserve space in advance if needed. File based requests get translated internally to a storage IO request. See section 6.14.1 for more information on file system support.

### 6.3.4.2.4.1 CsBlockIo

The data structure `CsBlockIo` is defined as follows:

```c
typedef struct {
    CS_STORAGE_IO_TYPE Type;             // see 6.3.4.1.7
    u32 StorageIndex;                    // denotes the index in a CSA, zero otherwise
    u32 NamespaceId;                     // represents a LUN or namespace
    u64 StartLba;
    u32 NumBlocks;
    CsDevAFDM DevMem;                    // see 6.3.4.2.3.1
} CsBlockIo;
```

### 6.3.4.2.4.2 CsFileIo

The data structure `CsFileIo` is defined as follows:

```c
typedef struct {
    CS_STORAGE_IO_TYPE Type;             // see 6.3.4.1.7
    void *FileHandle;
    u64 Offset;
    u32 Bytes;
    CsDevAFDM DevMem;                    // see 6.3.4.2.3.1
} CsFileIo;
```
6.3.4.2.4.3 CsStorageRequest

The data structure CsStorageRequest is defined as follows:

typedef struct {
    CS_STORAGE_REQ_MODE Mode;       // see 6.3.4.1.6
    CS_DEV_HANDLE DevHandle;         // the CSx handle
    union {
        CsBlockIo BlockIo;          // see Error! Reference source not found.
        CsFileIo FileIo;            // see Error! Reference source not found.
    } u;
} CsStorageRequest;

6.3.4.2.5 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.

6.3.4.2.5.1 CsComputeArg

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

typedef struct {
    enum CS_COMPUTE_ARG_TYPE Type;
    union {
        CsDevAFDM DevMem;              // see 6.3.4.2.3.1
        u64 Value64;
        u32 Value32;
        CS_STREAM_HANDLE StreamHandle;
    } u;
} CsComputeArg;

6.3.4.2.5.2 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 6.3.6
    int NumArgs;                    // set to total arguments to CSF
    CsComputeArg Args[1];           // see 6.3.4.2.5
    // allocate enough space past this for multiple arguments
} CsComputeRequest;
6.3.4.2.6 Batch Data Structures

Batch requests help optimize the total number of API requests by combining multiple requests into one batch. Batch requests also help execute repeatable tasks. Batch request setup is defined in detail under section Error! Reference source not found..

Each request in a batch is described by the CsBatchRequest data structure. The ReqType data field describes the type of request it holds under the union u, while reqLength denotes the size of the structure chosen. The reqLength is required for Compute operations as the request may contain more than one argument described in the Args field.

6.3.4.2.6.1 CsBatchRequest

The data structure CsBatchRequest is defined as follows:

```c
typedef struct {
    enum CS_BATCH_REQ_TYPE ReqType; // see 6.3.4.1.10
    u32 reqLength;
    union {
        CsCopyMemRequest CopyMem; // see Error! Reference source not found.
        CsStorageRequest StorageIo; // see Error! Reference source not found.
        CsComputeRequest Compute; // see 6.3.4.2.5.2
    } u;
} CsBatchRequest;
```

6.3.4.2.7 Statistics Data Structures

CSx statistics for specific resources may be queried using csQueryDeviceStatistics() API.

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 it. 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.

6.3.4.2.7.1 CSEUsage

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

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

6.3.4.2.7.2 CSxMemory

CSxMemory defines device memory usage.
All counters are represented in bytes if not specified.

```c
struct {
    u64 TotalAllocatedFDM;     // denotes the total FDM in bytes that have been allocated
    u64 LargestBlockAvailableFDM; // denotes the largest amount of FDM that may be allocated
    u64 AverageAllocatedSizeFDM; // denotes the average size of FDM allocations in bytes
    u64 TotalFreeCSFM;         // denotes the total CSFM 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;
```

6.3.4.2.7.3 CSFUsage

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

```c
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;
```

6.3.4.2.7.4 CsStatsInfo

The data structure CsStatsInfo is defined as follows:

```c
typedef union {
    CSEUsage CSEDetails;
    CSxMemoryUsage MemoryDetails;         // see 6.3.4.2.7.2
    CSFUsage CSFDetails;                  // see 6.3.4.2.7.3
} CsStatsInfo;
```

6.3.4.2.8 CSFIdInfo

The data structure CSFIdInfo is defined as follows:

```c
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
} CSFIdInfo;
```

6.3.4.2.9 CsCapabilities

The data structure CsCapabilities is defined as follows:

```c
typedef struct {
```
// specifies the fixed functionality device capability
struct {
    u64 Compression : 1;
    u64 Decompression : 1;
    u64 Encryption : 1;
    u64 Decryption : 1;
    u64 RAID : 1;
    u64 EC : 1;
    u64 Dedup : 1;
    u64 Hash : 1;
    u64 Checksum : 1;
    u64 RegEx : 1;
    u64 DbFilter : 1;
    u64 ImageEncode : 1;
    u64 VideoEncode : 1;
    u64 CustomType : 48;
} Functions;
} CsCapabilities;

Editor's note: Make an array of functionids

Editor's note: well know function definitions may be standardized. You may not need to discover capabilities of these. For pre-installed and downloaded CSFs there is a need for more information. This is currently a place holder.

Editor's note – add these to a CSEECapabilities:

    u64 ContainerImageLoader : 1;
    u64 eBPFEnvironment : 1;
    u64 FPGABitstreamLoader : 1;

6.3.4.2.10 **CsCapabilityInfo**

The data structure CsCapabilityInfo is defined as follows:

typedef union {
    // defines temperature details to set
    struct {
        s32 TemperatureLevel;
    } CSxTemperature;
    // defines CSx Max outstanding IOs allowed
    struct {
        u32 TotalOutstandingIOs;
    } MaxIOs;
} CsCapabilityInfo;

6.3.4.2.11 **CsDownloadInfo**

The data structure CsDownloadInfo is defined as follows:

typedef struct {
    enum CS_DOWNLOAD_TYPE Type;
    int SubType;                     // type dependent
    int Index;                       // program slot etc
    int Unload;                      // unload previously loaded entity
    int Length;                      // length in bytes of data in DataBuffer
} CsDownloadInfo;
void *DataBuffer;       // download data for program
}

Editor’s note: Needs to be updated to reflect CSEE download, activate, deactivate, unload changes in WG

Editor’s note: need function added to this struct

6.3.4.2.12 **CsQueryPluginRequest**

The data structure `CsQueryPluginRequest` is defined as follows:

```c
typedef struct {
    enum CS_PLUGIN_TYPE Type;   // see 6.3.4.1.15
    u32 InterfaceLength;
    u16 Id;
    union {
        TypeA; // TODO
        TypeB;
        TypeC;
    } Interface;
} CsQueryPluginRequest;
```

6.3.4.2.13 **CsPluginRequest**

The data structure `CsPluginRequest` is defined as follows:

```c
typedef struct {
    enum CS_PLUGIN_TYPE Type;   // see 6.3.4.1.15
    u32 InterfaceLength;
    u16 Id;
    union {
        TypeA; // TODO
        TypeB;
        TypeC;
    } Interface;
} CsPluginRequest;
```

6.3.5 **Resources**

**Table 10: Table of resources**

<table>
<thead>
<tr>
<th>Resource</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_DEV_HANDLE</td>
<td>The global device handle received back from <code>csOpenCSx</code></td>
</tr>
<tr>
<td>CS_CSE_HANDLE</td>
<td>The global device handle received back from <code>csOpenCSE</code></td>
</tr>
<tr>
<td>CS_MEM_HANDLE</td>
<td>Denotes a device memory handle and represents memory allocated on device</td>
</tr>
</tbody>
</table>
CS_CSF_ID  Denotes a computational storage function for all compute offload purposes
CS_EVT_HANDLE  Denotes an event handle for asynchronous IO
CS_BATCH_HANDLE  Denotes a batch request handle
CS_STREAM_HANDLE  Denotes a stream handle for streaming based compute offload

6.3.6 Resource Dependency
Table 10 describes the resource dependency for each resource.

![Resource dependency chart]

Figure 16: Resource dependency chart

Each resource created with the device is represented by a handle of type CS_XXX_HANDLE or CS_FUNCTION_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 are doing using the CS_FUNCTION_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 asynchronous mechanism may also be utilized with the callback option. Similarly, 
\texttt{CS\_MEM\_HANDLE} may be used by itself for device memory transfer operations.

The resource \texttt{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 is able to be kept unique when scaled.

6.3.7 Notification Callbacks

Common callback function definition to receive notifications on various CS based events. The callback is registered through the function \texttt{csRegisterNotify()}.

```c
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 the the CSE at the time it was opened.

Common callback function definition while queuing IO to the CSx

```c
typedef void(*csQueueCallbackFn)(void *QueueContext, CS_STATUS Status);
```
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 Length in bytes of buffer passed for output

OUT Buffer Returns a list of CSx names

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

If a valid Buffer pointer is specified where the length specified in Length is sufficient, then
it is updated with the list of CSx names available and Length updated to actual length of
string returned. If the length specified in Length is not sufficient to hold the contents
returned in Buffer, then Length will be populated with the required size and an error
status will be returned. An invalid input will return an error status.

If a NULL pointer is specified for Buffer and a valid pointer is provided for Length, then the
required buffer size is returned back in Length. The user will have to 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 CS_SUCCESS if there is no error and zero or more CSxes were
available for the list.

Otherwise, the function returns a status as of CS_INVALID_ARG or
CS_INVALID_LENGTH as defined in 6.3.2.

6.4.1.5 Notes
There may be one to multiple CSxes available on the system. The caller should always
check the value of Length in bytes for 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 each entry will be comma separated. The 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 `csOpenCSx()` to interface with the CSx.

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

```c
length = 0;
status = csQueryCSxList(&length, NULL);
if (status != CS_INVALID_LENGTH) {
    // return in error
}

csx_array = malloc(length);
status = csGetCSxList(&length, &csx_array[0]);
if (status == 0) {
    ...
}
```

### 6.4.2 `csQueryCSFList()`
This function returns zero or more CSFs available based on the query criteria.

#### 6.4.2.1 Synopsis

```c
CS_STATUS csQueryCSFList(char *Path, int *Length, char *Buffer);
```

#### 6.4.2.2 Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN Path</td>
<td>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.</td>
</tr>
<tr>
<td>IN OUT Length</td>
<td>Length of buffer passed for output</td>
</tr>
<tr>
<td>OUT Buffer</td>
<td>Returns a list of unique comma separated activated CSFs</td>
</tr>
</tbody>
</table>

#### 6.4.2.3 Description

`csQueryCSFList()` fills `Buffer` with a list of unique activated CSFs for the query based on `Path` if the length specified in `Length` is sufficient. The function may return one or more CSF names as a list in `Buffer` that match the `Path` criteria. Multiple CSFs when returned will be comma separated. A `Path` set to NULL is an invalid option and an error will be returned.
If a valid `Buffer` pointer is specified where the length specified in `Length` is sufficient, then it is updated with the list of CSF names available and `Length` updated to actual length of string 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` will be populated with the required size and an error status will be returned. An invalid input will return an error status.

If a null pointer is specified for `Buffer` and a valid pointer is provided for `Length`, then the required buffer size is returned back in `Length`. The user will have to 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.

### 6.4.2.4 Return Value

This function returns `CS_SUCCESS` if there is no error and one or more functions were available for the list.

Otherwise, the function returns a status of `CS_INVALID_ARG`, `CS_INVALID_LENGTH`, `CS_UNSUPPORTED`, `CS_OUT_OF_RESOURCES`, `CS_DEVICE_NOT_PRESENT`, `CS_ENTITY_NOT_ON_DEVICE`, `CS_ENODEV`, `CS_ENOENT` or `CS_ENXIO` as defined in 6.3.2.

### 6.4.2.5 Notes

If the `Path` input specified a device path or a CSx, then the CSF names returned if any shall be those available in that path. If the `Path` input specified a file or a directory, the query will reference 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` for 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 CSF entry, then each entry will be comma separated. This function may still return with success when `Length` is zero.

The returned list of CSF names 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

```c
CS_STATUS csGetCSxFromPath(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 Length of buffer passed for output

OUT DevName Returns the qualified name to the CSx

6.4.3.3 Description

csGetCSxFromPath() 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 Path, then all known CSxes are returned. Multiple CSxes when returned will be comma separated.

If a valid DevName buffer pointer is specified where the length specified in Length is sufficient, then it 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 will be populated with the required size and an error status will be returned. An invalid input returns an error status.

If a NULL pointer is specified for DevName and a valid pointer is provided for Length, then the requested buffer size is returned back in Length. The user may 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 CS_SUCCESS if there is no error and a CSx was found to be associated with the path specified.

Otherwise, the function returns a status of CS_INVALID_ARG, CS_ENOENT, CS_ENTITY_NOT_ON_DEVICE, CS_ENXIO, or CS_INVALID_LENGTH as defined in 6.3.2.

6.4.3.5 Notes

The Path parameter denotes the path to a device, filename or directory on a Linux filesystem. The path if partial will be 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. The function will return CS_ENXIO for such inputs.

The returned DevName is qualified to be used with the csOpenCSx() to interface with the CSE.
An example source fragment implementation would be:

```c
status = csGetCSxFromPath(my_file_with_path, &length, &csx_array[0]);
if (status == 0) {
    ...
}
```
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(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

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

If a valid DevName is specified, it is verified that it exists and is available and if all parameters are valid returns a handle to the CSx. An invalid input returns an error status.

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.

Otherwise, the function returns a status of CS_INVALID_ARG, CS_ENTITY_NOT_ON_DEVICE, or CS_NO_PERMISSIONS as defined in 6.3.2.

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 call. The CSx if open 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.

Otherwise, the function returns an error status of CS_INVALID_HANDLE as defined in 6.3.2.

6.5.3 csRegisterNotify()
Register a callback function to be notified based on various computational storage events across all CSxes and CSEs.

This is an optional function.

6.5.3.1 Synopsis
CS_STATUS csRegisterNotify(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, all CSEs and CSxes will be registered

IN NotifyOptions Denotes the notification types to registered to

IN NotifyFn A user specified callback notification function

6.5.3.3 Description
csRegisterNotify() registers the provided callback for notifications based on options selected in NotifyOptions by the user.

If a valid DevName is specified, the notifications will only be registered for the specified CSE or CSx. If NULL is specified, then the callback will be registered across all CSxes and CSEs. An invalid input returns an error status.
All input parameters are required for this function.

6.5.3.4 Return Value
This function returns CS_SUCCESS if there are no errors.

Otherwise, the function returns a status of CS_INVALID_ARG, CS_INVALID_OPTION, CS_DEVICE_NOT_PRESENT, CS_ENTITY_NOT_ON_DEVICE, CS_OUT_OF_RESOURCES or CS_NO_PERMISSIONS as defined in 6.3.2.

6.5.3.5 Notes
The callback is invoked by the API 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 on computational storage events. A callback function may have been previously registered using csRegisterNotify().

This is an optional function.

6.5.4.1 Synopsis
CS_STATUS csDeregisterNotify(char *DevName, csDevNotificationFn NotifyFn);

6.5.4.2 Parameters
IN DevName A string that denotes a specific CSE or CSx to deregister notifications from. If NULL, all CSEs and CSxes will be deregistered

IN NotifyFn The callback notification function previously registered

6.5.4.3 Description
csDeregisterNotify() removes a previously provided callback for notifications from one or more CSEs or CSxes.

If a valid DevName is specified, the notifications will only be deregistered for the specified CSE or CSx. If NULL, the callback will be deregistered across all CSxes and CSEs. An invalid input returns an error status.

All input parameters are required for this function.
6.5.4.4 Return Value
This function returns `CS_SUCCESS` if there are no errors.

Otherwise, the function returns a status of `CS_INVALID_ARG`, `CS_DEVICE_NOT_PRESENT`, `CS_ENTITY_NOT_ON_DEVICE`, `CS_OUT_OF_RESOURCES` or `CS_NO_PERMISSIONS` as defined in 6.3.2.

6.6 AFDM management

6.6.1 `csAllocMem()`
Allocates memory from the FDM for the requested size in bytes.

6.6.1.1 Synopsis

\[
\text{CS\_STATUS } \text{csAllocMem}((\text{CS\_DEV\_HANDLE } \text{DevHandle, int Bytes,)}
\]

\[
\text{unsigned int MemFlags, CS\_MEM\_HANDLE } \ast\text{MemHandle,}
\]

\[
\text{CS\_MEM\_PTR } \ast\text{VAddressPtr);}
\]

6.6.1.2 Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN DevHandle</td>
<td>Handle to CSx</td>
</tr>
<tr>
<td>IN Bytes</td>
<td>Length in bytes of FDM to allocate</td>
</tr>
<tr>
<td>IN MemFlags</td>
<td>Reserved, shall be zero</td>
</tr>
<tr>
<td>OUT MemHandle</td>
<td>Pointer to hold the memory handle once allocated</td>
</tr>
<tr>
<td>OUT VAddressPtr</td>
<td>Pointer 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</td>
</tr>
</tbody>
</table>

6.6.1.3 Description

`csAllocMem()` allocates requested memory from FDM.

If a valid `MemHandle` pointer is specified, it is updated with the handle to the AFDM. An invalid input returns an error status. If a valid `VAddressPtr` pointer is specified, the AFDM is mapped into the users virtual address space in host memory.

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.
Otherwise, the function returns an error status of CS_INVALID_HANDLE, CS_INVALID_ARG, CS_INVALID_OPTION, CS_NOT_ENOUGH_MEMORY, or CS_COULD_NOT_MAP_MEMORY as defined in 6.3.2.

6.6.1.5 Notes

AFDM is allocated using this call. AFDM is allocated on a host page size granularity and is rounded off for other values that are not in multiples of this size. It will be guaranteed that the virtual address pointer if requested will also be host page aligned.

The optional parameter VAddressPtr should only be used when the host application needs to transfer data 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 below

a) If the host application wants to use the direct p2p capability between Storage and AFDM, it provides VAddressPtr as the buffer to the filesystem to read or write to. Care should be taken while using this value that no buffering is enabled executing through a filesystem path. This is circumvented 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.

b) For usages where the host applications need to transfer data between host memory and device memory, this parameter is not required and should be set to NULL.

6.6.2 csFreeMem()

Frees AFDM for the memory handle specified.

6.6.2.1 Synopsis

CS_STATUS csFreeMem(CS_MEM_HANDLE MemHandle);

6.6.2.2 Parameters

IN MemHandle Handle to AFDM

6.6.2.3 Description

csFreeMem() frees previously requested AFDM.

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

All input parameters are required for this function.

6.6.2.4 Return Value

This function returns CS_SUCCESS if there is no error.
Otherwise, the function returns an error status of CS_INVALID_HANDLE, CS_UNKNOWN_MEMORY, CS_MEMORY_IN_USE, or CSDEVICE_NOT_AVAILABLE as defined in 6.3.2.

6.6.2.5 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.
6.7 Storage IOs

IO requests to and from storage devices are typically orchestrated through existing filesystems and block subsystem interfaces. To facilitate P2P between storage and CSxes, the AFDM should be allocated with virtual address mapping and this address pointer should be passed along to the filesystem/block subsystem. This allows the data to be loaded directly into AFDM from the application and directly into the host from the CSx.

For more advanced usages, P2P access alone may not be able to satisfy a user request. There will be usages where P2P does not work. This is work in progress.

a) the user does not want to deal with mapped virtual address buffers and file systems separately
b) the user wants the API library to also handle storage read/write requests for compute offload
c) the user provides the file handle and lets the library manage the storage access
d) the user provides a list of LBAs and device name/handle
e) the user requires remote CSxes

6.7.1 csQueueStorageRequest()

Queues a storage IO request to the device.

6.7.1.1 Synopsis

`CS_STATUS csQueueStorageRequest(CsStorageRequest *Req, void *Context, csQueueCallbackFn CallbackFn, CS_EVT_HANDLE EventHandle, u32 *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 needs 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 a valid value or if the request is synchronous.
Additional completion value provided as part of completion. This may be optional depending on the implementation.

### 6.7.1.3 Description

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

A valid `Req` structure (see Error! Reference source not found.) is required to initiate the storage IO operation. All fields in `Req` structure are required and describe the source and destinations details. To notify completion status, the inputs `CallbackFn` or `EventHandle` are required. Context is only required for `CallbackFn` or `EventHandle`. An `EventHandle` is utilized only by user space applications. Kernel space applications such as drivers and filesystems use the `CallbackFn`.

For `EventHandle`, see `csCreateEvent()` 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.

Otherwise, the function returns an error status of CS_INVALID_HANDLE, CS_INVALID_ARG, CS_INVALID_OPTION, CS_UNKNOWN_MEMORY, or CS_DEVICE_NOT_AVAILABLE as defined in 6.3.2.

### 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

```c
CS_STATUS csQueueCopyMemRequest(CsCopyMemRequest *CopyReq,
                                 void *Context, csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCallbackFn CallbackFn,
                                 csQueueCall...
CS_EVT_HANDLE EventHandle, u32 *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 needs 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 needs to be synchronous.

OUT CompValue Additional completion value provided as part of completion.
This may be optional depending on the implementation.

6.8.1.3 Description

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

A valid CopyReq structure (see Error! Reference source not found.) is required to initiate t
he copy operation. All fields in the CopyReq structure are required and describe the source
and destinations details. The inputs CallbackFn or EventHandle are required to notify
completion status. Context is only required for CallbackFn or EventHandle. See notes for
details.

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.

Otherwise, the function returns an error status of CS_INVALID_HANDLE,
CS_INVALID_ARG, CS_INVALID_OPTION, CS_UNKNOWN_MEMORY, or
CS_DEVICE_NOT_AVAILABLE as defined in 6.3.2.

Definitions
6.8.1.5 Notes

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

The `ByteOffset` field in `CsDevAFDM` may be set to zero for normal users. For advanced users, this field may be used in specifying one large device buffer with specific offsets for each request. One usage would be in a scatter gather list.

The copy operation may be requested to be synchronous or asynchronous. If synchronous, then all other inputs other than `CopyReq` must be set to NULL. If asynchronous, then either the `CallbackFn` or the `EventHandle` is required to be set to a valid value. If both are set it is treated as an error.

An example source fragment implementation to copy from host memory to device memory would be

```c
// copy 4kb from host buffer to offset 0 of device memory handle synchronously
copyReq.Type = CS_COPY_TO_DEVICE;
copyReq.HostMemPtr = &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);
if (status != CS_SUCCESS) {
    // handle error
    ...
}
```
6.9 CSF scheduling

CSxes provide functions in one or more device based functions to which compute work is able to be scheduled. These functions require a mechanism to invoke them and collect their results.

The following two functions provide the functionality to invoke one or more compute offload functions.

6.9.1 csGetCSFId()

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

6.9.1.1 Synopsis

```c
CS_STATUS csGetCSFId(CS_Dev_HANDLE DevHandle, char *CSFName, 
                     int *Length, CSFIdInfo *Buffer);
```

6.9.1.2 Parameters

- **IN DevHandle** Handle to CSx
- **IN CSFName** A pre-specified function name
- **IN OUT Length** The length of Buffer to hold CSFIdInfo details
- **OUT Buffer** A pointer to hold an array of CSFIdInfo data-structures with performance and power details of the CSF if successful

6.9.1.3 Description

`csGetCSFId()` returns one or more CSFIdInfo data-structures in Buffer when the length specified in Length is sufficient to satisfy the request. The CSFName should be a valid name that is available with the CSx as specified by DevHandle.

The function returns an error if the specified CSFName is not found.

If a valid Buffer pointer is specified where the length specified in Length is sufficient, then it is updated with an array of available CSFIdInfo data-structures and Length is updated to 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. An invalid input returns an error status.

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.
All input and output parameters are required for this function.

### 6.9.1.4 Return Value

`CS_SUCCESS` is returned if there are no errors in initializing the function.

Otherwise, the function returns an error status of `CS_INVALID_ARG`, `CS_INVALID_OPTION`, `CS_INVALID_HANDLE`, `CS_INVALID_CSF_NAME`, `CS_DEVICE_NOT_AVAILABLE` or as defined in 6.3.2.

### 6.9.1.5 Notes

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

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` and `RelativePower` data fields help differentiate between multiple CSF instances if received back from this function. The `Count` data field denotes the number of instances for this CSF and determines the parallelism available..

### 6.9.2 `csAbortCSF()`

Aborts the specified executing CSF if it is running any tasks.

#### 6.9.2.1 Synopsis

```
CS_STATUS csAbortCSF(CS_CSF_ID CSFId);
```

#### 6.9.2.2 Parameters

- **IN CSFId** The CSF to abort execution.

#### 6.9.2.3 Description

`csAbortCSF()` aborts the specified CSF from executing any further. Any outstanding request shall be 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.

Otherwise, the function returns an error status of `CS_INVALID_CSF_ID`, `CS_NO_PERMISSIONS`, `CS_FATAL_ERROR`, or `CS_DEVICE_NOT_AVAILABLE` as defined in 6.3.2.
6.9.2.5 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(CsComputeRequest *Req, void *Context, 
    csQueueCallbackFn CallbackFn, CS_EVT_HANDLE EventHandle, 
    u32 *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 needs 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 needs to be synchronous.

OUT CompValue Additional completion value provided as part of completion. This may be optional depending on the implementation.

6.9.3.3 Description
csQueueComputeRequest() queues a CSF request to the CSE.

6.9.3.4 Return Value
if there are no errors in, then for:

  a) a synchronous queue operation CS_SUCCESS is returned; and
  b) an asynchronous queue operation CS_QUEUED is returned.
Otherwise, the function returns an error status of CS_INVALID_ARG, CS_INVALID_OPTION, CS_INVALID_HANDLE, CS_UNKNOWN_MEMORY, or CS_DEVICE_NOT_AVAILABLE as defined in 6.3.2.

6.9.3.5 Notes
The CSF needs 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 6.3.4.2.5.2) provides inputs on the function the request should be issued to and its input arguments. The field NumArgs defines the number of arguments that need to be issued to the function. The user should ensure that these match actual function inputs. The runtime will not verify this.

See csQueueCopyMemRequest() (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 while kernel space applications such as drivers and filesystems use the CallbackFn.

For EventHandle, see csCreateEvent() for usage.

6.9.4 csHelperSetComputeArg()
Helper function that are able to optionally be used to set an argument for a compute request.

6.9.4.1 Synopsis
void csHelperSetComputeArg(CsComputeArg *ArgPtr, enum 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 may be one of the enum values.

IN <…> One or more variables that make up the argument by type.
6.9.4.2 Description

csHelperSetComputeArg() 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.

6.9.4.3 Return Value

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

6.9.4.4 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.

```c
// setup compute request with 3 arguments
req->DevHandle = devHandle;
req->FunctionId = functId;
req->NumArgs = 3;
argPtr = &req->Args[0];
csHelperSetComputeArg(&argPtr[0], CS_AFDM_TYPE, inMemHandle, 0);
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 process of scheduling batched requests helps in the following ways:

a) Minimize on host orchestration sub-tasks and associated latency costs;
b) Minimize on 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 APIs `csAllocBatchRequest()` (see 6.10.1), `csFreeBatchRequest()` (see 6.10.2), `csAddBatchEntry()` (see 6.10.3), `csHelperReconfigureBatchEntry()` (see 6.10.4), `csHelperResizeBatchRequest()` (see 6.10.5) and `csQueueBatchRequest()` (see 6.10.6). A batch operation is setup by first creating a batch request and then populating it with the list of requests. Once setup, the operation is able to be queued using the `csQueueBatchRequest()` API. 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 needed 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 need 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

```c
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

csAllocBatchRequest() 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.

Otherwise, the function returns an error status of CS_INVALID_ARG, CS_INVALID_OPTION, CS_OUT_OF_RESOURCES, or CS_DEVICE_NOT_AVAILABLE as defined in 6.3.2.

6.10.2 csFreeBatchRequest()

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

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

csFreeBatchRequest() 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 the freeing the batch resource.

Otherwise, the function returns an error status of CS_INVALID_ARG, CS_INVALID_HANDLE, or CS_NOT_DONE as defined in 6.3.2.

6.10.3 csAddBatchEntry()

Add a request to the batch request resource represented by the input handle. The request type is: storage, compute, or copy memory. Additionally, the batch index parameters places the request at the required point in the list of requests.
6.10.3.1 Synopsis

```
CS_STATUS csAddBatchEntry(CS_BATCH_HANDLE BatchHandle,
                          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 items that may contain more than one CSx based work items that may include storage requests, compute hardware functions and device memory copy requests.

**IN Req**
The request to add to the batch of requests represented by BatchHandle parameter. Denotes a compound request structure that describes the CSx batch items that contain the CSx based work item that may include storage request, compute hardware functions or compute memory copy requests.

**IN Before**
A batch entry index that denotes the position of an existing request entry that the current request will be inserted in front of. A zero value denotes the current request must be the first request. Any other non-zero value must represent a valid entry returned back a previous call to this function.

**IN After**
A batch entry index that denotes the position of an existing request entry that the current request will be inserted in after of. A zero value denotes the current request must be the first request. Any other non-zero value must represent a valid entry returned back a previous call to this function.

**OUT Curr**
A pointer to hold the output of the batch entry index for current request of successful.

6.10.3.3 Description

csAddBatchEntry() adds a request to a batch of requests represented by the BatchHandle parameter.

6.10.3.4 Return Value

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

Otherwise, the function returns an error status of CS_INVALID_ARG, CS_INVALID_OPTION, CS_INVALID_HANDLE, CS_UNKNOWN_MEMORY, CS_HANDLE_IN_USE, or CS_DEVICE_NOT_AVAILABLE as defined in 6.3.2.
Definitions

Notes

The parameter Req defines the individual requests themselves. It defines the type of batch request, namely \texttt{CS\_COPY\_DEV\_MEM}, \texttt{CS\_STORAGE\_IO} or \texttt{CS\_QUEUE\_COMPUTE} and the work item which may be either a data type of \texttt{CsCopyMemRequest}, \texttt{CsStorageRequest} or \texttt{CsComputeRequest}. See details under \texttt{csQueueCopyMemRequest()}, \texttt{csQueueStorageRequest()} and \texttt{csQueueComputeRequest()}.  

\textbf{6.10.4 csHelperReconfigureBatchEntry()}

Helps reconfigure an existing batch request entry with new request information.

\textbf{6.10.4.1 Synopsis}

\texttt{CS\_STATUS csHelperReconfigureBatchEntry(CS\_BATCH\_HANDLE BatchHandle,}
\texttt{CS\_BATCH\_INDEX Entry, CsBatchRequest *Req);}  

\textbf{6.10.4.2 Parameters}

- \textbf{IN BatchHandle} The handle previously allocated for batch requests.
- \textbf{IN Entry} The request’s batch entry index that is reconfigured.
- \textbf{IN Req} The new batch request entry details.

\textbf{6.10.4.3 Description}

\texttt{csHelperReconfigureBatchEntry()} Reconfigures an existing batch request entry located at the specified index denoted by \texttt{Entry} parameter.

\textbf{6.10.4.4 Return Value}

\texttt{CS\_SUCCESS} is returned if there are no errors in reconfiguring the batch request entry.

Otherwise, the function returns an error status of \texttt{CS\_INVALID\_ARG}, \texttt{CS\_INVALID\_OPTION}, \texttt{CS\_INVALID\_HANDLE}, or \texttt{CS\_UNKNOWN\_MEMORY} as defined in 6.3.2.

\textbf{6.10.5 csHelperResizeBatchRequest()}

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

```c
CS_STATUS csHelperResizeBatchRequest(CS_BATCH_HANDLE BatchHandle, int MaxReqs);
```

6.10.5.2 Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN BatchHandle</td>
<td>The handle previously allocated for batch requests that is resized.</td>
</tr>
<tr>
<td>IN MaxReqs</td>
<td>The maximum number of requests the caller perceives that this batch resource is resized to. The parameter may not exceed the maximum supported by the CSE.</td>
</tr>
</tbody>
</table>

6.10.5.3 Description

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

6.10.5.4 Return Value

`CS_SUCCESS` is returned if there are no errors in the resizing of the resource. Otherwise, the function returns an error status of `CS_INVALID_ARG`, `CS_INVALID_OPTION`, `CS_OUT_OF_RESOURCES`, or `CS_DEVICE_NOT_AVAILABLE` as defined in 6.3.2.

6.10.6 `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 must already have been populated with the list of batched requests.

6.10.6.1 Synopsis

```c
CS_STATUS csQueueBatchRequest(CS_BATCH_HANDLE BatchHandle, 
                              void *Context, csQueueCallbackFn CallbackFn, 
                              CS_EVT_HANDLE EventHandle, u32 *CompValue);
```

6.10.6.2 Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN BatchHandle</td>
<td>The handle previously allocated for batch requests.</td>
</tr>
</tbody>
</table>
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 needs to be asynchronous.

IN EventHandle A handle to an event previously created using cscCreateEvent(). This value may be NULL if CallbackFn parameter is specified to be valid value or if also set to NULL when the request needs to be synchronous.

OUT CompValue Additional completion value provided as part of completion. This may be optional depending on the implementation.

6.10.6.3 Description

csQueueBatchRequest() queues a batch of requests that is able to include complex flows through storage for P2P, compute and data copies with the CSE.

6.10.6.4 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.

Otherwise, the function returns an error status of CS_INVALID_ARG, CS_INVALID_OPTION, CS_INVALID_HANDLE, CS_UNKNOWN_MEMORY, or CS_DEVICE_NOT_AVAILABLE as defined in 6.3.2.

6.10.6.5 Notes

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

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) Chain the output of the first CSF to a second CSF and so forth;
f) Load storage data and metadata in parallel and 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 must first allocate a batch handle using `csAllocBatchRequest()` and then add individual requests using `csAddBatchEntry()`.

Batching requests require the `Mode` field input which may be `CS_BATCH_SERIAL`, `CS_BATCH_PARALLEL` and `CS_BATCH_HYBRID`. This input instructs the runtime on how to handle this request. Serialized requests are those that depend on the previous requests output as their input. Parallelized requests are breaking down multiple requests into smaller requests that execute all 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. Do note that any device memory used in these requests by `CS_MEM_HANDLE` type is only allowed to be utilized on the device it was allocated on. 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 must 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 IO request that needs 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 will look up dependency of memory resources in the list. Optimizations on queuing requests independently versus after the dependency are taken based on this information presented by the order of the list of requests in the batch. With `CS_BATCH_HYBRID` mode, complex flow graphs are able to be processed where multiple serial and parallel flows are possible to be accommodated. Additional details on this usage is provided under hybrid operations in section 5.2.1.3.

The requirements on the `CallbackFn` and `EventHandle` apply the same way as in `csQueueCopyMemRequest()`. An `EventHandle` will be utilized only by user space applications while function space users such as drivers and filesystems will use the `CallbackFn`.

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

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

```c
// 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);
allocate storage, compute and DMA requests and set them up..
status = csAddBatchEntry(BatchHandle, &storReq, 0, 0, &storEntry);
status = csAddBatchEntry(BatchHandle, &compReq, 0, storEntry, &compEntry);
status = csAddBatchEntry(BatchHandle, &copyReq, 0, compEntry, &copyEntry);
process through entire data file of 1GB
while (fileSize) {
    status = csQueueBatchRequest(BatchHandle, &myContext, NULL, NULL, NULL);
    fileSize -= dataSize;
    advance file pointer to next 1MB
    storReq.u.StorageIo.u.FileIo.Offset += dataSize;
    status = csHelperReconfigureBatchEntry(BatchHandle, storEntry, &storReq);
}
status = csFreeBatchRequest(BatchHandle);
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 hold the event handle once allocated

6.11.1.3 Description
csCreateEvent() allocates and initializes a system event resource.
If a valid EventHandle pointer is specified, it is updated with the handle to the allocated event resource. An invalid input will return an error status.
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.
Otherwise, the function returns an error status of CS_INVALID_ARG or CS_NOT_ENOUGH_MEMORY as defined in 6.3.2.

6.11.1.5 Notes
Event resource is not allocated at the device level but rather at the system level. It may be used with any CSx. Once used, it will be referenced by that device and hence 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 that needs to be freed
6.11.2.3 Description

csDeleteEvent() deletes a previously allocated event resource with csCreateEvent().

If a valid EventHandle is specified, it is freed and returned back to the system. An invalid input will return an error status.

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.

Otherwise, the function returns an error status of CS_INVALID_HANDLE as defined in 6.3.2.

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);

6.11.3.2 Parameters

IN EventHandle The event handle that needs to be polled

OUT Context The context to the event that completed

6.11.3.3 Description

csPollEvent() queries a previously allocated event resource with csCreateEvent() when used with CSFs. The Context parameter returned will refer to the original context provided when the request was made.

If a valid EventHandle is specified, it is queried for any pending events. An invalid input will return an error status.

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.
Otherwise, the function returns an error status of CS_INVALID_HANDLE, CS_DEVICE_NOT_AVAILABLE, CS_DEVICE_ERROR, CS_FATAL_ERROR, or CS_ERROR_IN_EXECUTION as defined in 6.3.2 that maps to the work item it was included in.

6.11.3.5 Notes
An event resource is submitted to csQueueCopyMemRequest(), csQueueComputeRequest(), or csQueueBatchRequest() for any polling. It is the responsibility of the user to ensure that the correct event handle was used to poll and it was not freed use csDeleteEvent().
6.12 Management

Device management provides functions that are used to query and manage the device properties and resources.

6.12.1 csQueryDeviceProperties()

Queries the CSx for its properties.

This is a privileged function.

6.12.1.1 Synopsis

\[
\text{CS\_STATUS} \quad \text{csQueryDeviceProperties}(\text{CS\_DEV\_HANDLE} \; \text{DevHandle}, \\
\text{CS\_RESOURCE\_TYPE} \; \text{Type}, \; \text{int} \; *\text{Length}, \; \text{CsProperties} \; *\text{Buffer});
\]

6.12.1.2 Parameters

- **IN** DevHandle: Handle to CSx
- **IN** Type: The type of CSx resource to query
- **IN OUT** Length: Length in bytes of buffer passed for output
- **OUT** Buffer: A pointer to a buffer that is able to hold all the device properties.

6.12.1.3 Description

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

If a valid Buffer pointer where the length specified in Length is sufficient, then it 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 will be populated with the required size and an error status will be returned.

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

If a valid pointer is specified for Buffer and 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 back in Length.
All input parameters are required for this function.

6.12.1.4 Return Value

cs_success is returned if there are no errors.

Otherwise, the function returns an error status of CS_INVALID_ARG, CS_INVALID_HANDLE, CS_INVALID_LENGTH, CS_NOT_ENOUGH_MEMORY, or CS_DEVICE_NOT_AVAILABLE as defined in 6.3.2.

6.12.1.5 Notes

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

A user will utilize this function early on in device setup and verify that the properties are as expected and proceed to configure the CSx. Refer to Error! Reference source not found. for the data structure details and their individual fields.

6.12.2 csQueryDeviceCapabilities()

Queries the CSE for its capabilities. These capabilities may be computational storage related functions that are built-in.

6.12.2.1 Synopsis

CS_STATUS csQueryDeviceCapabilities(CS_DEV_HANDLE DevHandle,
CsCapabilities *Caps);

6.12.2.2 Parameters

IN DevHandle Handle to CSx

IN Caps A pointer to a buffer that is able to hold all the CSx capabilities

6.12.2.3 Description

csQueryDeviceCapabilities() returns the device capabilities that are built-in.

All input parameters are required for this function.

6.12.2.4 Return Value

cs_success is returned if there are no errors.

Otherwise, the function returns an error status of CS_INVALID_ARG, CS_INVALID_HANDLE, CS_NOT_ENOUGH_MEMORY, or CS_DEVICE_NOT_AVAILABLE as defined in 6.3.2.
6.12.2.5 Notes
The capabilities returned provide information on built-in fixed functions. These may be as defined by NVMe Computational Storage.

A user on a successful query would probe Caps.Functions first and find out if a specific function of interest is available. If true, the data fields under Caps.Function.FunctionType should be reviewed for a match.

6.12.3 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.

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

6.12.3.2 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 that will hold the requested CSE statistics

6.12.3.3 Description
csQueryDeviceStatistics() 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 is optional and is required only for certain statistics types. The Identifier is used with structures CSEDetails and CSFDetails. When used for CSEDetails, the Identifier field 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 shall be set to its unique CSEId available in csQueryDeviceProperties(). Similarly, for specific CSF statistics, the Identifier must be set to its unique CSFId also available through csQueryDeviceProperties(). An error is returned if the Identifier is set to NULL while Type requires it.
All input parameters are required for this function.

6.12.3.4 Return Value

cs_success is returned if there are no errors.

Otherwise, the function returns an error status of CS_INVALID_ARG, CS_INVALID_HANDLE, CS_NO_PERMISSIONS, CS_NOT_ENOUGH_MEMORY, or CS_DEVICE_NOT_AVAILABLE as defined in 6.3.2.

6.12.3.5 Notes

The Statistics returned provide information on CSX usage like utilization and health. Some of the statistics reflected will be preserved since the power on state. The counters will not be reset on a query.

6.12.4 csSetDeviceCapability()

Set the CSXes specific capability. A specific capability setting is able to be changed by the requested type.

This is a privileged function.

6.12.4.1 Synopsis

CS_STATUS csSetDeviceCapability(CS_DEV_HANDLE DevHandle,

CS_CAP_TYPE Type, CsCapabilityInfo *Details);

6.12.4.2 Parameters

IN DevHandle Handle to CSx

IN Type Capability type to set

IN Details A pointer to a structure that holds the capability details to set

6.12.4.3 Description

csSetDeviceCapability() sets the device capability requested based on Type.

All input parameters are required for this function.

6.12.4.4 Return Value

cs_success is returned if there are no errors.

Otherwise, the function returns an error status of CS_INVALID_ARG, CS_INVALID_HANDLE, CS_NOT_ENOUGH_MEMORY, CS_DEVICE_NOT_AVAILABLE, or CS_NO_PERMISSIONS as defined in 6.3.2.
6.12.4.5 Notes
The CsCapabilitiesInfo structure provides the input information to modify the CSE. This information is able to be manage specific capabilities exposed by the device.

This is a privileged user operation. This user privilege has different definitions based on OS type but primarily involves a super user/administrator with rights to change device configuration that a typical user is not allowed to perform.

6.12.5 csDownload()
Downloads a specified CSF to a CSE that is programmable. A function may also be downloadable that may contain one or more CSFs. It is implementation specific as to how the downloaded code is secured.

This is a privileged function.

6.12.5.1 Synopsis
CS_STATUS csDownload(CS_DEV_HANDLE DevHandle,
CsDownloadInfo *ProgramInfo);

6.12.5.2 Parameters
IN DevHandle Handle to CSE
IN Info A pointer to a buffer that holds the program details to download

6.12.5.3 Description
csDownload() downloads a program to a reprogrammable CSE with the suggested program details if successful. The CsDownloadInfo structure shall provide the details of downloaded contents such as the program’s type. Program types that may be downloaded are dependent on the CSE’s support. A user can query the CSx for what it supports through csQueryDeviceCapabilities().

All input parameters are required for this function.

6.12.5.4 Return Value
CS_SUCCESS is returned if there are no errors.

Otherwise, the function returns an error status of CS_INVALID_ARG, CS_UNSUPPORTED, CS_INVALID_HANDLE, CS_NO_PERMISSIONS< CS_LOAD_ERROR, CS_NOT_ENOUGH_MEMORY, or CS_DEVICE_NOT_AVAILABLE as defined in 6.3.2.
6.12.5.5 Notes
Only CSxes that contain a CSE that is capable of processing a CSF accept this call. CSx devices that only have fixed functionality fail this call.

6.12.6 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 API. Prior to usage, these resources have to be activated.

This is a privileged function.

6.12.6.1 Synopsis
CS_STATUS csConfig(CS_Dev_HANDLE DevHandle, int *Length, CsConfigInfo *Info, CsConfigData *Data);

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

6.12.6.3 Description
csConfig() 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.

6.12.6.4 Return Value
CS_SUCCESS is returned if there are no errors.

Otherwise, the function returns an error status of CS_INVALID_ARG, CS_UNSUPPORTED, CS_INVALID_HANDLE, CS_NO_PERMISSIONS, CS_LOAD_ERROR, CS_NOT_ENOUGH_MEMORY or CS_DEVICE_NOT_AVAILABLE as defined in 6.3.2.
6.12.6.5 Notes

Only CSxes that contain a CSE that is capable of processing configuration input accept this call. Refer to section Error! Reference source not found. for more details on the data structures and sub-structures details.
6.12.7 csAbortCSE()
Aborts all outstanding and queued transactions to the CSE.
This is a privileged function.

6.12.7.1 Synopsis
CS_STATUS csAbortCSE(CS_CSE_HANDLE CSEHandle);

6.12.7.2 Parameters
IN CSEHandle Handle to CSE

6.12.7.3 Description
csAbortCSE() will abort all outstanding transactions to the CSE and dequeue all I/Os queued if successful.
All input parameters are required for this function.

6.12.7.4 Return Value
CS_SUCCESS is returned if there are no errors.
Otherwise, the function returns an error status of CS_UNSUPPORTED, CS_INVALID_HANDLE, CS_NOT_ENOUGH_MEMORY, or CS_DEVICE_NOT_AVAILABLE as defined in 6.3.2.

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

6.12.8 csResetCSE()
Resets the CSE.
This is a privileged function.

6.12.8.1 Synopsis
CS_STATUS csResetCSE(CS_CSE_HANDLE CSEHandle);

6.12.8.2 Parameters
IN CSEHandle Handle to CSE
6.12.8.3 Description

`csResetCSE()` resets the CSE if successful. As part of resetting, all outstanding transactions to the CSE are aborted and all IOs are de-queued.

All input parameters are required for this function.

6.12.8.4 Return Value

`CS_SUCCESS` is returned if there are no errors.

Otherwise, the function returns an error status of `CS_UNSUPPORTED`, `CS_INVALID_HANDLE`, `CS_NOT_ENOUGH_MEMORY`, or `CSDEVICE_NOT_AVAILABLE` as defined in 6.3.2.

6.12.8.5 Notes

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

6.13 Stream Management

6.13.1 `csAllocStream()`

Allocates a stream resource with the device.

6.13.1.1 Synopsis

```c
CS_STATUS csAllocStream(CS_DEV_HANDLE DevHandle,
                         CS_STREAM_TYPE Type,
                         CS_STREAM_HANDLE *StreamHandle);
```

Editor's note: There is no definition of `CS_STREAM_HANDLE`. Also need a model section for streams – Stephen B.

6.13.1.2 Parameters

- **IN DevHandle**
  - Handle to CSx

- **IN Type**
  - The type of stream to allocate. This parameter is currently reserved or not in use.

- **OUT StreamHandle**
  - A pointer to a buffer to hold the returned stream handle if successful

6.13.1.3 Description

`csAllocStream()` reserves a storage stream from the device.

All parameters are required for this function.
6.13.1.4 Return Value

`CS_SUCCESS` is returned if there are no errors.

Otherwise, the function returns an error status of `CS_UNSUPPORTED`, `CS_INVALID_HANDLE`, `CS_OUT_OF_RESOURCES`, or `CS_DEVICE_NOT_AVAILABLE` as defined in 6.3.2.

6.13.1.5 Notes

A stream must be first allocated before using it. The device may support only a few number of streams and may/may not advertise it.

6.13.2 `csFreeStream()`

Releases a previously allocated stream resource with the device.

6.13.2.1 Synopsis

```c
CS_STATUS csFreeStream(CS_STREAM_HANDLE StreamHandle);
```

Editor's note: `StreamHandle` is not unique across CSxes so a `CS_DEV_HANDLE` is required here.

Parameters

- `IN StreamHandle` A stream handle that was previously allocated with `csAllocStream()` request

6.13.2.2 Description

`csFreeStream()` releases a stream back to the device.

All parameters are required for this function.

6.13.2.3 Return Value

`CS_SUCCESS` is returned if there are no errors.

Otherwise, the function returns an error status of `CS_INVALID_HANDLE`, `CS_NOT_ENOUGH_MEMORY`, or `CS_DEVICE_NOT_AVAILABLE` as defined in 6.3.2.

6.13.2.4 Notes

A stream must have been allocated to release it. The device may support only a few number of streams and may/may not advertise it.
6.14 Library Management

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

Editor’s note: This is a work in progress and needs additional specification

6.14.1 csQueryLibrarySupport()

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

6.14.1.1 Synopsis

\[
\text{CS\_STATUS csQueryLibrarySupport}(\text{enum CS\_LIBRARY\_SUPPORT Type, int *Length, char *Buffer});
\]

6.14.1.2 Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN Type</td>
<td>Library support type query</td>
</tr>
<tr>
<td>IN OUT Length</td>
<td>Length of buffer passed for output</td>
</tr>
<tr>
<td>OUT Buffer</td>
<td>Returns a list of CSEs</td>
</tr>
</tbody>
</table>

6.14.1.3 Description

\text{csQueryLibrarySupport()} 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 will be a set of strings separated by commas. Currently, only file-system support is able to be queried.

If a valid Buffer pointer is specified where the length specified in Length is sufficient, then it is updated with the list of all items that match support for Type and Length updated to actual length of string. If the length specified in Length is not sufficient to hold the contents returned in Buffer, then Length will be populated with the required size and an error status will be returned. An invalid input will return an error status.

If a NULL pointer is specified for Buffer and a valid pointer is provided for Length, then the required buffer size is returned back in Length. The user will have to 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.
6.14.1.4 Return Value

CS_SUCCESS if there is no error and the query for Type was met.

Otherwise, the function returns an error status of CS_INVALID_ARG or CS_INVALID_LENGTH as defined in 6.3.2.

6.14.1.5 Notes

The caller should always check the value of Length for 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. The function may still return with success when Length is zero.

The returned queried list is able to be parsed and verified as the user intended. For example, if the user wanted to find out if the library supports file systems of specific types, the input would specify that for Type and the output would contain a list of file systems that the library is able to support for storage requests.

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

```c
length = 0;
status = csQueryLibrarySupport(CS_FILE_SYSTEMS_SUPPORTED, &length, NULL);
if (status != CS_INVALID_LENGTH) {
    // return in error
}
fs_list = malloc(length);
status = csGetCSxList(CS_FILE_SYSTEMS_SUPPORTED, &length, &fs_list[0]);
if (status == 0) {
    ...
}
```

6.14.2 csQueryPlugin()

Queries the API library for registered plugins.

This is a privileged function.

6.14.2.1 Synopsis

```c
CS_STATUS csQueryPlugin(CsQueryPluginRequest *Req,
                            csQueryPluginCallbackFn CallbackFn);
```

6.14.2.2 Parameters

IN Req Request structure for type of plugins to query
IN CallbackFn  Callback function to call into when requested query is satisfied

6.14.2.3  Description

`csQueryPlugin()` if successful calls into the callback function provided when it matches the query.

All input parameters are required for this function.

6.14.2.4  Return Value

`CS_SUCCESS` is returned if there are no errors.

Otherwise, the function returns an error status of `CS_INVALID_ARG`, `CS_INVALID_OPTION`, or `CS_NOT_ENOUGH_MEMORY` as defined in 6.3.2.

The callback function is defined as follows:

```c
typedef void(*csQueryPluginCallbackFn)(enum CS_PLUGIN_TYPE Type, char *Buffer);
```

6.14.2.5  Notes

This functionality is used by a privileged process to query the current registered plugins in the system. Computational storage device providers and vendors who provide their own plugin support would be interested in this call.

The callback function will be called when the required query is met. The callback may be called multiple times if more than one plugin type was queried upon.

6.14.3  `csRegisterPlugin()`

Registers a specified plugin with the API library.

This is a privileged function.

6.14.3.1  Synopsis

```c
CS_STATUS csRegisterPlugin(CsPluginRequest *Req);
```

6.14.3.2  Parameters

IN Req  Request structure to register a plugin

6.14.3.3  Description

`csRegisterPlugin()` registers the specified plugin if successful.
All input parameters are required for this function.

### 6.14.3.4 Return Value

`CS_SUCCESS` is returned if there are no errors.

Otherwise, the function returns an error status of `CS_INVALID_ARG`, `CS_INVALID_OPTION`, or `CS_NOT_ENOUGH_MEMORY` as defined in 6.3.2.

### 6.14.3.5 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 be interested in this call.

### 6.14.4 `csDeregisterPlugin()`

Deregisters a specified plugin from the API library.

This is a privileged function.

#### 6.14.4.1 Synopsis

```c
CS_STATUS csDeregisterPlugin(CsPluginRequest *Req);
```

#### 6.14.4.2 Parameters

- **IN Req** Request structure to deregister a plugin

#### 6.14.4.3 Description

`csDeregisterPlugin()` will deregister the specified plugin if successful.

All input parameters are required for this function.

#### 6.14.4.4 Return Value

`CS_SUCCESS` is returned if there are no errors.

Otherwise, the function returns an error status of `CS_INVALID_ARG`, `CS_INVALID_OPTION`, or `CS_NOT_ENOUGH_MEMORY` as defined in 6.3.2.

#### 6.14.4.5 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 would be interested in this call.
A Sample Code

A.1 Initialization and queuing a synchronous request

A synchronous (blocking) request where the user waits for the IO to complete is illustrated in the following example.

Initialization may occur in the following way.

// discover my device
length = sizeof(csxBuff);
status = csGetCSxFromPath("myFileToAccelerate", &length, &csxBuff);
if (status != CS_SUCCESS)
    ERROR_OUT("No CSx device found\n");
// open device, init function and prealloc buffers
status = csOpenCSx(csxBuff, &MyDevContext, &dev);
if (status != CS_SUCCESS)
    ERROR_OUT("Could not access device\n");
// query device properties & capabilities
status = csGetCSFId(dev, "decrypt", &infoLength, &csfInfo);
if (status != CS_SUCCESS)
    ERROR_OUT("CSX does not contain any decrypt CSFs \n");
// pick CSF
decryptId = findMyCSF(&csfInfo, size);
// allocate device and host memory
for (i = 0; i < 2; i++) {
    status = csAllocMem(dev, CHUNK_SIZE, 0, &AFDMArray[i]);
    if (status != CS_SUCCESS)
        ERROR_OUT("AFDM alloc error\n");
}
Source data may be fetched in the following way.

// copy encrypted data from host memory into AFDM
// allocate copy request and issue it
copyReq = calloc(1, sizeof(CsCopyMemRequest));
if (!copyReq)
    ERROR_OUT("request alloc error\n");
// setup copy request
    copyReq->Type = CS_COPY_TO_DEVICE;
    copyReq->HostVAddress = encrypt_buf;
    copyReq->DevMem.MemHandle = AFDMArray[0];
copyReq->DevMem.ByteOffset = 0;
copyReq->Bytes = CHUNK_SIZE;

// issue a synchronous copy request
status = csQueueCopyMemRequest(copyReq, NULL, NULL, NULL, NULL);
if (status != CS_SUCCESS)
  ERROR_OUT("Copy to AFDM error\n");

Compute execution may occur in the following way.

// allocate compute request for 3 args
req = calloc(1, sizeof(CsComputeRequest) + (sizeof(CsComputeArg) * 3));
if (!req)
  ERROR_OUT("request alloc error\n");

// setup work request
req->CSFId = decryptId;
req->NumArgs = 3;
argPtr = &req->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(req, 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. The following code snippet demonstrates the changes to compute execution while applying an event based mechanism.

// allocate event for async processing
status = csCreateEvent(&evtHandle[0]);
if (status != CS_SUCCESS)
  ERROR_OUT("Could not create event\n");

// allocate compute request for 3 args
req = calloc(1, sizeof(CsComputeRequest) + (sizeof(CsComputeArg) * 3));
if (!req)
  ERROR_OUT("request alloc error\n");

// setup work request
req->CSFId = decryptId;
req->NumArgs = 3;
argPtr = &req->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 asynchronous compute request
status = csQueueComputeRequest(req, req, NULL, evtHandle[0], NULL);
if (status != CS_SUCCESS)
    ERROR_OUT("Compute exec error\n");
while ((status = csPollEvent(evtHandle[0], &context)) != CS_SUCCESS) {
    // IO not done; do other work
}

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

// do asynchronous work request
status = csQueueComputeRequest(req, req, MyAsyncFn, NULL, NULL);
if (status != CS_SUCCESS)
    ERROR_OUT("Compute exec error\n");
// IO not done; do other work till callback is invoked in separate thread context

A.3 Using Batch processing

Batch processing aids is processing more than one request optimally as one request from the call. `csQueueBatchRequest()` is able to take multiple requests as one request as specified in section 6.100. The following example illustrates a sequence of serialized batch processing requests. Data is first read from the storage device and populated in AFDM. In the second request, the CSF is run on the data read to decompress it into another AFDM buffer. In the third and last request, the contents of the second buffer are copied into host memory. The batch of requests are set to execute serially and have no dependencies except for serialization which is handled by this batch type. The request is set to execute asynchronously in non-blocking mode.

// batch execute storage IO + 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 IO. Batch only for LBA based IO
// for others use normal file IO not with batch
storReq = calloc(1, sizeof(CsBatchRequest));
if (!storReq)
    ERROR_OUT("memory alloc error\n");
storReq->reqType = CS_STORAGE_IO;
storReq->DevHandle = devHandle;
storReq->u.StorageIo.Mode = CS_STORAGE_BLOCK_IO;
storReq->u.StorageIo.u.BlockIo.Type = CS_STORAGE_LOAD_TYPE;
storReq->u.StorageIo.u.BlockIo.StartLba = LBAs[0];
storReq->u.StorageIo.u.BlockIo.NumBlocks = 1;
storReq->u.StorageIo.u.BlockIo.DevMemMemHandle = inMemHandle;
storReq->u.StorageIo.u.BlockIo.DevMemByteOffset = 0;
status = csAddBatchEntry(batchHandle, storReq, 0, 0, &storEntry);
if (status != CS_SUCCESS)
    ERROR_OUT("batch request error\n");
// next setup compute IO with 3 CSF arguments
computeReq = calloc(1, sizeof(CsBatchRequest) + (sizeof(CsComputeArg) * 3));
if (!computeReq)
    ERROR_OUT("memory alloc error\n");
computeReq->reqType = CS_QUEUE_COMPUTE;
computeReq->u.Compute.DevHandle = devHandle;
computeReq->u.Compute.FunctionId = funcId;
computeReq->u.Compute.NumArgs = 3;
argPtr = &computeReq->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 = csAddBatchEntry(batchHandle, computeReq, 0, storEntry, &computeEntry);
if (status != CS_SUCCESS)
    ERROR_OUT("batch request error\n");
// lastly setup DMA results to host
copyReq = calloc(1, 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.HostVAddress = resBuffer;
copyReq->u.CopyMem.DevMemMemHandle = outMemHandle;
copyReq->u.CopyMem.DevMem.ByteOffset = 0;
copyReq->u.CopyMem.Bytes = 4096 * 3;
status = csAddBatchEntry(batchHandle, copyReq, 0, computeEntry, &copyEntry);
if (status != CS_SUCCESS)
    ERROR_OUT("batch request error\n");
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 the previous input completion is able to be waited upon to start the next request. The example reads data from storage, runs parallel compute offload operation on it 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.

```c
// hybrid batch setup execution
// large storage IO + 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 IO. Batch only LBA based IO
// for others use normal file IO not with batch
storReq = calloc(1, 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.u.BlockIo.Type = CS_STORAGE_LOAD_TYPE;
storReq->u.StorageIo.u.BlockIo.StartLba = LBAs[0];
storReq->u.StorageIo.u.BlockIo.NumBlocks = 32;
storReq->u.StorageIo.u.BlockIo.DevMem.MemHandle = inMemHandle;
storReq->u.StorageIo.u.BlockIo.DevMem.ByteOffset = 0;
status = csAddBatchEntry(batchHandle, storReq, 0, 0, &storEntry);
if (status != CS_SUCCESS)
    ERROR_OUT("batch request error\n");  ```
// allocate memory for parallel compute batch requests and reuse req
computeReq = calloc(1, sizeof(CsBatchRequest) + (sizeof(CsComputeArg) * 3));
if (!computeReq)
    ERROR_OUT("memory alloc error\n");
inMemOffset = 0;
for (i = 0; i < 8; i++) {
    // next setup compute IO with 3 arguments each
    computeReq->reqType = CS_QUEUE_COMPUTE;
    computeReq->u.Compute.FunctionId = csfId;
    computeReq->u.Compute.NumArgs = 3;
    argPtr = &computeReq->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 = csAddBatchEntry(batchHandle, computeReq, 0, storEntry, &computeEntryArray[i]);
    if (status != CS_SUCCESS)
        ERROR_OUT("batch request error\n");
inMemOffset += 16384;
}
// now allocate memory for parallel DMA batch requests and reuse req
copyReq = calloc(1, 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.HostVAddress = &resBuffer[outMemOffset];
    copyReq->u.CopyMem.DevMem.ByteOffset = 0;
    copyReq->u.CopyMem.Bytes = 4096;
    status = csAddBatchEntry(batchHandle, copyReq, 0, computeEntryArray[j],
                              &copyEntryArray[j]);
    if (status != CS_SUCCESS)
        ERROR_OUT("batch request error\n");
    outMemOffset += 4096;
}
// all done, queue the batch request
status = csQueueBatchRequest(batchHandle, NULL, NULL, evtHandle, NULL);
if (status != CS_SUCCESS)
    ERROR_OUT("batch request error\n");
// wait on the final results
while ((status = csPollEvent(evtHandle, &context)) != CS_SUCCESS) {
    // IO not done; do other work
    // poll for previous IOs too and mark them done
}

A.5 Using files for storage IO

Using the filesystem managed files for reading and writing data is a powerful interface
that the API csQueueStorageRequest() 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 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 API as shown
below. Here 128K bytes are read from storage using the file handle and loaded in
AFDM.

    // query capabilities for file IO 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 IO request for file usage
storReq = calloc(1, 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;
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);
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) {
    // IO not done; do other work
}