Computational Storage API
Version 0.5 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 9, 2021
USAGE
Copyright © 2021 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) 2021, 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>0.0r2</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>0.0r3</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>0.0r4</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>0.0r5</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>0.0r6</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>0.0r7</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>0.0r8</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>Version</td>
<td>Date</td>
<td>Author</td>
<td>Changes</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>------------</td>
<td>------------------</td>
<td>-------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>0.0r9</td>
<td>5/27/2021</td>
<td>Bill Martin</td>
<td>Incorporated Oscar Pinto/Bill Martin CSE/CSx proposal approved on 5/25</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Moved all unresolved comments to editor’s notes</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>One exception is to add language about adding text on insufficient length into all appropriate APIs. This will be done prior to release.</td>
<td></td>
</tr>
<tr>
<td>0.0r10</td>
<td>6/1/2021</td>
<td>Bill Martin, Oscar Pinto</td>
<td>Added changes to how length is handled if too short</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Changes proposed by Oscar Pinto</td>
<td></td>
</tr>
<tr>
<td>0.0r11</td>
<td>6/3/2021</td>
<td>Bill Martin</td>
<td>Removed change tracking</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Updated references and added a few missing cross references</td>
<td></td>
</tr>
<tr>
<td>0.5r0</td>
<td>6/9/2021</td>
<td>Arnold Jones</td>
<td>Formatted as a Working Draft for release outside the TWG per TWG approval ballot.</td>
<td></td>
</tr>
</tbody>
</table>

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

1 SCOPE ............................................................................................................................................. 13
  1.1 ABOUT COMPUTATIONAL STORAGE APIs ................................................................. 13
  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 .............................................................................................................. 18
    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>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.19</td>
<td>NVMe®</td>
<td>18</td>
</tr>
<tr>
<td>2.1.20</td>
<td>Peer-to-Peer</td>
<td>18</td>
</tr>
<tr>
<td>2.1.21</td>
<td>P2P</td>
<td>18</td>
</tr>
<tr>
<td>2.1.22</td>
<td>PCIe®</td>
<td>19</td>
</tr>
<tr>
<td>2.1.23</td>
<td>Virtual Machine</td>
<td>19</td>
</tr>
<tr>
<td>2.2</td>
<td>KEYWORDS</td>
<td>19</td>
</tr>
<tr>
<td>2.2.1</td>
<td>mandatory</td>
<td>19</td>
</tr>
<tr>
<td>2.2.2</td>
<td>may</td>
<td>19</td>
</tr>
<tr>
<td>2.2.3</td>
<td>may not</td>
<td>19</td>
</tr>
<tr>
<td>2.2.4</td>
<td>need not</td>
<td>19</td>
</tr>
<tr>
<td>2.2.5</td>
<td>optional</td>
<td>19</td>
</tr>
<tr>
<td>2.2.6</td>
<td>shall</td>
<td>19</td>
</tr>
<tr>
<td>2.2.7</td>
<td>should</td>
<td>20</td>
</tr>
<tr>
<td>2.3</td>
<td>ABBREVIATIONS</td>
<td>20</td>
</tr>
<tr>
<td>2.4</td>
<td>REFERENCES</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>COMPUTATIONAL STORAGE</td>
<td>21</td>
</tr>
<tr>
<td>4</td>
<td>APIS OVERVIEW</td>
<td>24</td>
</tr>
<tr>
<td>4.1</td>
<td>DISCOVERY &amp; DIFFERENCES IN CONNECTIVITY</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>CONFIGURATION</td>
<td>30</td>
</tr>
<tr>
<td>4.3</td>
<td>FDM ALLOCATION</td>
<td>30</td>
</tr>
<tr>
<td>4.4</td>
<td>COMPUTE TYPES AND EXECUTION</td>
<td>30</td>
</tr>
<tr>
<td>4.5</td>
<td>DOWNLOADING FUNCTIONS</td>
<td>30</td>
</tr>
<tr>
<td>4.6</td>
<td>EXTENDING API SUPPORT</td>
<td>31</td>
</tr>
<tr>
<td>4.7</td>
<td>ASSOCIATION OF CSP AND STORAGE</td>
<td>31</td>
</tr>
<tr>
<td>4.8</td>
<td>API USAGE EXAMPLE</td>
<td>31</td>
</tr>
</tbody>
</table>
5 DETAILS ON COMMON USAGES.............................................................................33
5.1 FDM Usage ..................................................................................................33
  5.1.1 FDM usage example for CSD ..............................................................33
  5.1.2 Allocating from FDM............................................................................34
  5.1.3 FDM to host memory mapping............................................................35
  5.1.4 Copy data between host memory and AFDM .....................................37
5.2 SCHEDULING COMPUTE OFFLOAD JOBS..............................................38
  5.2.1 Batching requests ..............................................................................40
  5.2.2 Optimal Scheduling...........................................................................44
5.3 GROUPING CSFs ..........................................................................................46
5.4 COMPLETION MODELS ..............................................................................47

6 CS API INTERFACE DEFINITIONS .............................................................48
6.1 API ACCESS AND FLOW CONVENTIONS ...........................................48
6.2 USAGE OVERVIEW ......................................................................................49
6.3 COMMON DEFINITIONS .............................................................................53
  6.3.1 Data Types ..........................................................................................53
  6.3.2 Status Values ......................................................................................53
  6.3.3 Notification Options ..........................................................................56
  6.3.4 Data Structures ...................................................................................56
  6.3.5 Resources .........................................................................................65
  6.3.6 Resource Dependency .......................................................................66
  6.3.7 Notification Callbacks .......................................................................67
6.4 DISCOVERY ..................................................................................................68
  6.4.1 csQueryCSEList() .............................................................................68
  6.4.2 csQueryFunctionList() ......................................................................69
<table>
<thead>
<tr>
<th>Section</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4.3</td>
<td>csGetCSxFromPath()</td>
<td>71</td>
</tr>
<tr>
<td>6.5</td>
<td>ACCESS</td>
<td>73</td>
</tr>
<tr>
<td>6.5.1</td>
<td>csOpenCSx()</td>
<td>73</td>
</tr>
<tr>
<td>6.5.2</td>
<td>csCloseCSx()</td>
<td>73</td>
</tr>
<tr>
<td>6.5.3</td>
<td>csGetCSEFromCSx()</td>
<td>74</td>
</tr>
<tr>
<td>6.5.4</td>
<td>csOpenCSE()</td>
<td>75</td>
</tr>
<tr>
<td>6.5.5</td>
<td>csCloseCSE()</td>
<td>76</td>
</tr>
<tr>
<td>6.5.6</td>
<td>csRegisterNotify()</td>
<td>76</td>
</tr>
<tr>
<td>6.5.7</td>
<td>csDeregisterNotify()</td>
<td>77</td>
</tr>
<tr>
<td>6.6</td>
<td>AFDM MANAGEMENT</td>
<td>78</td>
</tr>
<tr>
<td>6.6.1</td>
<td>csAllocMem()</td>
<td>78</td>
</tr>
<tr>
<td>6.6.2</td>
<td>csFreeMem()</td>
<td>79</td>
</tr>
<tr>
<td>6.7</td>
<td>STORAGE IOS</td>
<td>81</td>
</tr>
<tr>
<td>6.7.1</td>
<td>csQueueStorageRequest()</td>
<td>81</td>
</tr>
<tr>
<td>6.8</td>
<td>CSx DATA MOVEMENT</td>
<td>82</td>
</tr>
<tr>
<td>6.8.1</td>
<td>csQueueCopyMemRequest()</td>
<td>82</td>
</tr>
<tr>
<td>6.9</td>
<td>CSF SCHEDULING</td>
<td>85</td>
</tr>
<tr>
<td>6.9.1</td>
<td>csGetFunction()</td>
<td>85</td>
</tr>
<tr>
<td>6.9.2</td>
<td>csStopFunction()</td>
<td>86</td>
</tr>
<tr>
<td>6.9.3</td>
<td>csQueueComputeRequest()</td>
<td>86</td>
</tr>
<tr>
<td>6.9.4</td>
<td>csHelperSetComputeArg()</td>
<td>88</td>
</tr>
<tr>
<td>6.10</td>
<td>BATCH SCHEDULING</td>
<td>90</td>
</tr>
<tr>
<td>6.10.1</td>
<td>csAllocBatchRequest()</td>
<td>90</td>
</tr>
<tr>
<td>6.10.2</td>
<td>csFreeBatchRequest()</td>
<td>91</td>
</tr>
<tr>
<td>6.10.3</td>
<td>csAddBatchEntry()</td>
<td>91</td>
</tr>
</tbody>
</table>
6.10.4 csHelperReconfigureBatchEntry() .................................................. 93
6.10.5 csHelperResizeBatchRequest() ...................................................... 93
6.10.6 csQueueBatchRequest() .................................................................. 94

6.11 EVENT MANAGEMENT ........................................................................ 98
6.11.1 csCreateEvent() ............................................................................ 98
6.11.2 csDeleteEvent() ............................................................................. 98
6.11.3 csPollEvent() .................................................................................. 99

6.12 MANAGEMENT .................................................................................... 101
6.12.1 csQueryDeviceForComputeList() .................................................... 101
6.12.2 csQueryDeviceProperties() ............................................................ 102
6.12.3 csQueryDeviceCapabilities() .......................................................... 103
6.12.4 csQueryDeviceStatistics() ............................................................... 104
6.12.5 csSetDeviceCapability() ................................................................. 105
6.12.6 csDownload() ................................................................................. 106
6.12.7 csConfig() ..................................................................................... 107
6.12.8 csAbortCSE() ................................................................................ 109
6.12.9 csResetCSE() ................................................................................ 109

6.13 STREAM MANAGEMENT .................................................................... 110
6.13.1 csAllocStream() .......................................................................... 110
6.13.2 csFreeStream() ............................................................................. 111

6.14 LIBRARY MANAGEMENT .................................................................. 112
6.14.1 csQueryLibrarySupport() ............................................................... 112
6.14.2 csQueryPlugin() ........................................................................... 113
6.14.3 csRegisterPlugin() ...................................................................... 114
6.14.4 csDeregisterPlugin() .................................................................... 115
A | SAMPLE CODE.......................................................................................... 116
   | A.1 Initialization and queuing a synchronous request.......................... 116
   | A.2 Queuing an asynchronous request................................................... 117
   | A.3 Using batch processing...................................................................... 118
   | A.4 Applying hybrid batch processing feature......................................... 120
   | A.5 Using files for storage IO................................................................. 122
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>An Architectural view of Computational Storage</td>
<td>22</td>
</tr>
<tr>
<td>Figure 2</td>
<td>API interrelationships</td>
<td>26</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Example API flows</td>
<td>32</td>
</tr>
<tr>
<td>Figure 4</td>
<td>System Memory Map</td>
<td>37</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Example data transfers between AFDM in a CSx and host memory</td>
<td>38</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Batch requests</td>
<td>40</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Optimal CSF Scheduling</td>
<td>45</td>
</tr>
<tr>
<td>Figure 8</td>
<td>CSF Grouping</td>
<td>46</td>
</tr>
<tr>
<td>Figure 9</td>
<td>API access flows</td>
<td>49</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Resource dependency chart</td>
<td>66</td>
</tr>
</tbody>
</table>
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:

- Additional definitions to support a device managed interface
- CsComputeRequest should be enhanced to be able to operate directly with storage not just AFDM
- Include a diagram that shows the application, API, library, and CSx (including the existing diagram of a CSx)
- Provide a mechanism to avoid polling for results and provides real-time output results

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., `csQueryCSEList`) and provided to the `csOpenCSE` function

   Editor’s Note: Need to define how CSE name is assigned and how is it set? How is it Globally Unique within environment?
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 Programming 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
CSA Computational Storage Array
CSD Computational Storage Drive
CSE Computational Storage Engine
CSF Computational Storage Function
CSFM Computational Storage Function Memory
CSP Computational Storage Processor
CSx Computational Storage devices
DMA Direct Memory Access
FDM Function Data Memory
FPGA Field-Programmable Gate Array
NVM Non-Volatile Memory
P2P Peer-to-Peer
PM Persistent Memory
SGL Scatter Gather List
SSD Solid State Disk
VM Virtual Machine

2.4 **References**

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

SNIA Computational Storage 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**

<table>
<thead>
<tr>
<th>Example of CSFs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compression</td>
</tr>
<tr>
<td>Encryption</td>
</tr>
<tr>
<td>Database filter</td>
</tr>
<tr>
<td>Erasure coding</td>
</tr>
<tr>
<td>RAID</td>
</tr>
<tr>
<td>Hash/CRC</td>
</tr>
<tr>
<td>RegEx (pattern matching)</td>
</tr>
<tr>
<td>Scatter Gather</td>
</tr>
<tr>
<td>Pipeline</td>
</tr>
<tr>
<td>Video compression</td>
</tr>
<tr>
<td>Data Deduplication</td>
</tr>
<tr>
<td>Large Data Set</td>
</tr>
</tbody>
</table>

**Table 2: Example Computational Storage Engine Environment Types**

<table>
<thead>
<tr>
<th>Enumerated Image types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System Image Loader</td>
</tr>
</tbody>
</table>

**Figure 1: An Architectural view of Computational Storage**
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. Additional libraries are able to be built on this base set of functions. This version of the standard is tailored towards 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.

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 2.
Figure 2: API interrelationships
4.1 Discovery & differences in connectivity
As shown in
Figure 1, computational storage is provided by CSxes (i.e., CSDs, CSPs and CSAs). Each of these may have their own configurations, that may be specified prior to use (see 6.12.7). 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.

The APIs provide an interface through which discovery and configuration may be accomplished using plugins (see 4.13).

There may be a mix of device types and interfaces to discover that increases the complexity of the discovery APIs. However, device vendors are able to implement specific logic in plugins which are outside the scope of this specification and interface to the discovery process of the APIs. The API may discover properties which are not changeable and capabilities that may be changed by the application. Device vendors may implement specific logic in plugins and interface to the discovery process of the APIs. Once discovered, this API provides an abstraction for discovering and using the functions provided by those CSxes:

a) `csQueryCSEList()`, an API that provides a list of all known CSE;
b) `csGetCSEFromPath()`, provides the CSE path from a device or filename path;
c) `csQueryDeviceProperties()`, helps discover device details;


d) **csQueryDeviceCapabilities()**, helps discover additional capabilities that the device supports.

### 4.2 Configuration

Configuration may be conducted at the CSx, CSE, CSEE and CSF level. Configuration may involve setting up device parameters and configuration details that may be defined or vendor specific. The `csConfig()` API helps setup configuration for these various entities. Configuration may also involve a download operation which may be applied by the `csDownload()` API.

In addition to the configuration, the `csSetDeviceCapability()` provides the mechanisms to configure the device capabilities.

### 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.

Editor's Note: Need to add a description of how plugins work.

4.8 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 3 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 3: 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 `csAllocMem()` and `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 `csAllocMem()` API.

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

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

Data transfers between AFDM and host memory are conducted using the `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 architecture’s 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 DMA’ed 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 3 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 4, 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;
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

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:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Reduction of PCIe bus bandwidth utilization;</td>
</tr>
<tr>
<td>b</td>
<td>Reduction in CPU utilization due to reduced memory copies; and</td>
</tr>
<tr>
<td>c</td>
<td>Reduction in host memory utilization.</td>
</tr>
</tbody>
</table>

### 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 5.
The job request. Compute offload jobs require input and produce output. Each of these entities require a
cs
An advanced method of queuing jobs is batching multiple ex
their values should match the definition of the offload function as the runtime will not
input the
Scheduling compute offload is done with
QueueBatchRequest
QueueComputeRequest
Figure 5: 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 3 summarizes job processing for input, compute and output.
### Table 3: Job request processing

<table>
<thead>
<tr>
<th>Job</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</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>
</tbody>
</table>
| Storage     | a) Use file system calls with device memory mapped to host; and  
b) Use csQueueStorageRequest() with type option CS_STORAGE_LOAD_TYPE |   |
| Host memory | Use csQueueCopyMemRequest() or csQueueBatchRequest() with option CS_COPY_TO_DEVICE                                                   |   |
| Compute     | The actual compute job may be scheduled to run in the following ways:                                                                     |
|             | **method** | **Related function**                                                                                                                             |
| Single or batch request | Use csQueueComputeRequest() for a single request or csQueueBatchRequest() for batch request.                                      |   |
| Output      | Provides output from a compute job. Output from a compute job may be received in two ways:                                               |
|             | **Input method** | **Related function**                                                                                                                                 |
| Storage     | a) Use file system calls with device memory mapped to host; and  
b) Use csQueueStorageRequest() with type option CS_STORAGE_STORE_TYPE |   |
| Host memory | csQueueCopyMemRequest() or csQueueBatchRequest() with option CS_COPY_FROM_DEVICE                                                       |   |
5.2.1 Batching 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 6: Batch requests**

Figure 6 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 3.

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.

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.

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.

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

In the following example, the output of a compute request becomes the input to the next compute request.
5.2.1.2 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’s 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 6 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 `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 `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 Mode specifying the operation for that node.

Table 4 summarizes batch operations.
### Table 4: 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 of CSF scheduling](image)

**Figure 7: Optimal CSF Scheduling**

Figure 7 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 CSE’s 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 8: CSF Grouping

Figure 8 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

Scheduling compute tasks, storage requests and copying data from AFDM is able to be setup for synchronous or asynchronous completions. This is specified as part of the input parameters to the requests and summarized in Table 5.

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

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

Table 6: 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>csQueryCSELList()</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>- Identify CSEs</td>
<td>csQueryFunctionList()</td>
</tr>
<tr>
<td>- Identify CSE associated with Storage device</td>
<td>csGetCSxFromPath()</td>
</tr>
<tr>
<td></td>
<td>csGetCSEFromCSx()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Device Access</th>
<th>csOpenCSx()</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Open/Close CSE device for access</td>
<td>csCloseCSx()</td>
</tr>
<tr>
<td></td>
<td>csOpenCSE()</td>
</tr>
<tr>
<td></td>
<td>csCloseCSE()</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>CSE Data movement</th>
<th>csQueueCopyMemRequest()</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Transfer data from/to device memory to/from host memory</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CSE function and scheduling</th>
<th>csGetFunction()</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Initialize CSE(s)</td>
<td>csStopFunction()</td>
</tr>
<tr>
<td>- Schedule CSF on device</td>
<td>csQueueComputeRequest()</td>
</tr>
<tr>
<td></td>
<td>csHelperSetComputeArg()</td>
</tr>
<tr>
<td></td>
<td>csQueueBatchRequest()</td>
</tr>
<tr>
<td></td>
<td>csAllocBatchRequest()</td>
</tr>
<tr>
<td></td>
<td>csFreeBatchRequest()</td>
</tr>
<tr>
<td></td>
<td>csAddBatchEntry()</td>
</tr>
<tr>
<td></td>
<td>csHelperReconfigureBatchEntry()</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>csHelperResizeBatchRequest()</td>
<td></td>
</tr>
<tr>
<td>csQueryDeviceProperties()</td>
<td>Query device properties and capabilities</td>
</tr>
<tr>
<td>csQueryDeviceCapabilities()</td>
<td></td>
</tr>
<tr>
<td>csQueryDeviceStatistics()</td>
<td>Manage device functionality</td>
</tr>
<tr>
<td>csQueryDeviceForComputeList()</td>
<td></td>
</tr>
<tr>
<td>csGroupComputeByIds()</td>
<td></td>
</tr>
<tr>
<td>csUngroupComputeFromGroupId()</td>
<td></td>
</tr>
<tr>
<td>csSetDeviceCapability()</td>
<td></td>
</tr>
<tr>
<td>csDownload()</td>
<td></td>
</tr>
<tr>
<td>csConfig()</td>
<td></td>
</tr>
<tr>
<td>csAbortCSE()</td>
<td></td>
</tr>
<tr>
<td>csResetCSE()</td>
<td></td>
</tr>
<tr>
<td>csRegisterNotify()</td>
<td></td>
</tr>
<tr>
<td>csDeregisterNotify()</td>
<td></td>
</tr>
<tr>
<td>csCreateEvent()</td>
<td>Event Management</td>
</tr>
<tr>
<td>csDeleteEvent()</td>
<td>Create/delete events for completion processing</td>
</tr>
<tr>
<td>csPollEvent()</td>
<td></td>
</tr>
<tr>
<td>csAllocStream()</td>
<td>Stream Management</td>
</tr>
<tr>
<td>csFreeStream()</td>
<td>Allocate/free streams for stream based processing</td>
</tr>
<tr>
<td>csQueryLibrarySupport()</td>
<td>Library Management</td>
</tr>
<tr>
<td>csQueryPlugin()</td>
<td>Query API library support</td>
</tr>
<tr>
<td>csRegisterPlugin()</td>
<td>Manage library interfaces to support APIs</td>
</tr>
<tr>
<td>csDeregisterPlugin()</td>
<td></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 7 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>CSDEVICE_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>CSENTITY_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>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CS_INVALID_ID</td>
<td>The specified input ID was invalid and does not exist</td>
</tr>
<tr>
<td>CS_INVALID_LENGTH</td>
<td>The specified buffer is not of sufficient length</td>
</tr>
<tr>
<td>CS_INVALID_OPTION</td>
<td>An invalid option was specified</td>
</tr>
<tr>
<td>CS_INVALID_FUNCTION</td>
<td>The function specified was invalid</td>
</tr>
<tr>
<td>CS_INVALID_FUNCTION_NAME</td>
<td>The function name specified does not exist or is invalid</td>
</tr>
<tr>
<td>CS_IO_TIMEOUT</td>
<td>An IO submitted has timed out</td>
</tr>
<tr>
<td>CS_LOAD_ERROR</td>
<td>The specified download could not be initialized</td>
</tr>
<tr>
<td>CS_MEMORY_IN_USE</td>
<td>The requested memory is still in use</td>
</tr>
<tr>
<td>CS_NO_PERMISSIONS</td>
<td>There were insufficient permissions to proceed with request</td>
</tr>
<tr>
<td>CS_NOT_DONE</td>
<td>The request is not done</td>
</tr>
<tr>
<td>CS_NOT_ENOUGH_MEMORY</td>
<td>There is not enough memory to satisfy the request</td>
</tr>
<tr>
<td>CS_NO_SUCH_ENTITY_EXISTS</td>
<td>There is no such entry that exists</td>
</tr>
<tr>
<td>CS_OUT_OF_RESOURCES</td>
<td>The system is out of resources to satisfy the request</td>
</tr>
<tr>
<td>CS_QUEUEED</td>
<td>The request was successfully queued</td>
</tr>
<tr>
<td>CS_UNKNOWN_MEMORY</td>
<td>The memory referenced was unknown</td>
</tr>
<tr>
<td>CS_UNKNOWN_COMPUTE_FUNCTION</td>
<td>The function referenced is unknown</td>
</tr>
<tr>
<td>CS_UNSUPPORTED</td>
<td>The request is not supported</td>
</tr>
</tbody>
</table>
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 8: 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_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_NOTIFYRESOURCE_WARNING</td>
<td>The CSx is running out of resources</td>
</tr>
<tr>
<td>CS_NOTIFYDOWNLOAD_INFO</td>
<td>Additional information available for downloaded program</td>
</tr>
<tr>
<td>CS_NOTIFYCONFIG_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_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_DOWNLOAD_TYPE;

Editor’s Note: Work on download and config to reflect: CSF/CSEE/VS; CSF: FBGA_bitstream; eBPF; Container; VS;
6.3.4.1.2 **CS CONFIG TYPE**

typedef enum {
    CS_CSF_TYPE = 1,
    CS_VENDOR_SPECIFIC = 2
} CS_CONFIG_TYPE;

Editor's Note: Add CSEE type

6.3.4.1.3 **CS MEM COPY TYPE**

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

6.3.4.1.4 **CS STORAGE REQ Mode**

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

6.3.4.1.5 **CS STORAGE IO TYPE**

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

6.3.4.1.6 **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;

6.3.4.1.7 **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;

6.3.4.1.8 **CS BATCH REQ TYPE**

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

6.3.4.1.9 **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

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;

6.3.4.1.10 **CS CAP TYPE**

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

    Editor's note: These need further work

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

6.3.4.1.11 **CS STREAM TYPE**

The data type CS_STREAM_TYPE is defined as follows:

typedef enum {
    CS_STREAM_COMPUTE_TYPE = 1
} CS_STREAM_TYPE;

6.3.4.1.12 **CS LIBRARY SUPPORT**

    Editor's note: These need further work
typedef enum {
    CS_FILE_SYSTEMS_SUPPORTED = 1,
    // TODO:
    CS_RESERVED = 2
} CS_LIBRARY_SUPPORT;

6.3.4.1.13  **CS PLUGIN TYPE**

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

6.3.4.1.14  **Properties data structures**

The data structure CSxProperties includes the properties of the CSx and one or more instances of CSEProperties as defined below.

The structure CSxProperties provides information pertaining to the CSx and are fixed attributes that do not change while it is functional. The FDMIsDeviceManaged field when set to 1 identifies that the CSx manages FDM for allocations and deallocations and shall decide how the memory is managed. The FDMIsHostVisible 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.

The sub-structure CSEProperties provides information on the specific CSE. 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 MaxFunctionParametersAllowed field denotes the maximum parameters supported by the CSE. A function cannot exceed this number and will be rejected if it does by the queueing API. The MaxConcurrentFunctionInstances field denotes how many instances of the same function can be run concurrently at any given time. Some CSE's may support concurrency of functions if their hardware supports it. The device may choose to delay or reject requests if they exceed this setting.

6.3.4.1.14.1  CSEProperties

typedef struct {
    u16 HwVersion;
    u16 SwVersion;
    char UniqueName[32];  // an identifiable string for this CSE
    u16 NumBuiltInFunctions;  // number of available preloaded functions
    u32 MaxRequestsPerBatch;  // maximum number of requests supported per batch request
    u32 MaxFunctionParametersAllowed;  // maximum number of parameters supported
    u32 MaxConcurrentFunctionInstances;  // maximum number of function instances supported
} CSEProperties;
6.3.4.14.2 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
    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;
    u16 NumCSEs;
    CSEProperties CSE[1];      // see 6.3.4.14
} CSxProperties;

6.3.4.2 Structures

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

6.3.4.2.1 CsDevAFDM

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

6.3.4.2.2 CsBlockIo

typedef struct {
    CS_STORAGE_IO_TYPE Type;       // see 6.3.4.1.5
    u32 StorageIndex;              // denotes the index in a CSA, zero otherwise
    u32 NamespaceId;               // represents a LUN or namespace
    u64 StartLba;
    u32 NumBlocks;
    CsDevAFDM DevMem;              // see 6.3.4.2.1
} CsBlockIo;

6.3.4.2.3 CsFileIo

typedef struct {
    CS_STORAGE_IO_TYPE Type;       // see 6.3.4.1.5
    Void *FileHandle;
    u64 Offset;
    u32 Bytes;
    CsDevAFDM DevMem;              // see 6.3.4.2.1
} CsFileIo;
6.3.4.2.4 **CsStorageRequest**

The structure **CsStorageRequest** describes the storage IO request between the storage device and the CSF. Storage IO is able to be block or file for a single request and utilizes the `Mode` field to select it. The `Type` field describes the direction of data flow from storage device.

Block requests describe details such as the namespace to operate on, the LBA and number of blocks to transfer. They also describe the AFDM that the transfer occurs to/from. The `StorageIndex` field specifies the drive to target the request to in a CSA and is reserved for other CSx types. See `csDevAFDM()` for `DevMem` field details as specified in section 6.8.1.

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.

```c
typedef struct {
    CS_STORAGE_REQ_MODE Mode; // see 6.3.4.1.4
    CS_DEV_HANDLE DevHandle;
    union {
        CsBlockIo BlockIo; // see 6.3.4.2.2
        CsFileIo FileIo;  // see 6.3.4.2.3
    } u;
} CsStorageRequest;
```

6.3.4.2.5 **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.

```c
typedef struct {
    enum CS_MEM_COPY_TYPE Type; // see 6.3.4.1.3
    void *HostVAddress;
    CsDevAFDM DevMem; // see 6.3.4.2.1
    unsigned int Bytes;
} CsCopyMemRequest;
```

6.3.4.2.6 **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 6.3.4.2.1
        u64 Value64;
        u32 Value32;
        CS_STREAM_HANDLE StreamHandle;
    } u;
} CsComputeArg;

6.3.4.2.7 CsComputeRequest

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

typedef struct {
    CS_CSE_HANDLE CSEHandle;
    CS_FUNCTION_ID FunctionId;
    int NumArgs;        // set to total arguments to CSF
    CsComputeArg Args[1];    // allocate enough space past this for multiple arguments
} CsComputeRequest;

Editor's note: Need a definition for FunctionId; How we abstract this needs to be determined. Need a mechanism for vendorID/FunctionID.

6.3.4.2.8 CsBatchRequest

typedef struct {
    enum CS_BATCH_REQ_TYPE ReqType;    // see 6.3.4.1.8
    u32 reqLength;
    union {
        CsCopyMemRequest CopyMem;    // see 6.3.4.2.5
        CsStorageRequest StorageIo;  // see 6.3.4.2.4
        CsComputeRequest Compute;    // see 6.3.4.2.7
    } u;
} CsBatchRequest;

6.3.4.2.9 CsFunctionInfo

The data structure CsFunctionInfo is defined as follows:

typedef struct {
    CS_FUNCTION_ID FunctionId;
    u8 NumUnits;  //number of instances of this function available
    char Name[32];
} CsFunctionInfo;

6.3.4.2.10 CsCapabilities

The data structure CsCapabilities is defined as follows:

typedef struct {
    // specifies the fixed functionality device capability
    struct {
    // content of the structure
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 function ids

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 eBPFLoader : 1;
u64 FPGABitstreamLoader : 1;

6.3.4.2.11 CSEUsage

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

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

6.3.4.2.12 CSxMemory

CSxMemory defines device memory usage.

All counters are represented in bytes if not specified.

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 TotalFDtHostinMB;  // total FDM transferred to host memory in megabytes
    u64 TotalHosttoFDMinMB;  // total host memory transferred to FDM in megabytes
} CSx_memory;
6.3.4.2.13 **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 function in seconds
    u64 TotalExecutions; // number of executions performed
    u64 ShortestTimeUsecs; // the shortest time the function ran in microseconds
    u64 LongestTimeUsecs; // the longest time the function ran in microseconds
    u64 AverageTimeUsecs; // the average runtime in microseconds
} CSFUsage;
```

6.3.4.2.14 **CsStatsInfo**

The data structure CsStatsInfo is defined as follows:

```c
typedef union {
    CSEUsage CSEDetails;
    CSxMemoryUsage MemoryDetails; // see 6.3.4.2.12
    CSFUsage FunctionDetails; // see 6.3.4.2.13
} CsStatsInfo;
```

6.3.4.2.15 **CsCapabilityInfo**

The data structure CsCapabilityInfo is defined as follows:

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

6.3.4.2.16 **CsConfigInfo**

The data structure CsConfigInfo is defined as follows:

```c
typedef struct {
    enum CS_CONFIG_TYPE Type; // type dependent
    int SubType;
    int Index; // program slot etc if applicable
    int Length; // length in bytes of data in DataBuffer
    void *DataBuffer; // configuration data to download
} CsConfigInfo;
```
6.3.4.2.17 **CsDownloadInfo**

The data structure CsDownloadInfo is defined as follows:

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

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

6.3.4.2.18 **CsQueryPluginRequest**

The data structure CsQueryPluginRequest is defined as follows:

typedef struct {
    enum CS_PLUGIN_TYPE Type;         // see 6.3.4.1.13
    u32 InterfaceLength;              // TODO
    u16 Id;
    union {
        TypeA;                         // TODO
        TypeB;
        TypeC;
    } Interface;
} CsQueryPluginRequest;

6.3.4.2.19 **CsPluginRequest**

The data structure CsPluginRequest is defined as follows:

typedef struct {
    enum CS_PLUGIN_TYPE Type;         // see 6.3.4.1.13
    u32 InterfaceLength;              // TODO
    u16 Id;
    union {
        TypeA;                         // TODO
        TypeB;
        TypeC;
    } Interface;
} CsPluginRequest;

6.3.5 **Resources**

Table 9: Table of resources

<table>
<thead>
<tr>
<th>Resource</th>
<th>Details</th>
</tr>
</thead>
</table>
CS_DEV_HANDLE  The global device handle received back from \texttt{csOpenCSx}

CS_CSE_HANDLE  The global device handle received back from \texttt{csOpenCSE}

CS_MEM_HANDLE  Denotes a device memory handle and represents memory allocated on device

CS_FUNCTION_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 9 describes the resource dependency for each resource.

![Resource Dependency Chart](image)

**Figure 10: 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, `CS_MEM_HANDLE` may be used by itself for device memory transfer operations.

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

This function returns one or more CSEs available based on the query criteria.

6.4.1.1 Synopsis

\[
\text{CS\_STATUS csQueryCSEList(char *FunctionName, int *Length, char *Buffer);} \\
\]

Editor's note: FunctionName should be FunctionId

Editor's note: Make this on a per CSx basis with a null CSx handle indicating all CSEs on all CSxes

6.4.1.2 Parameters

IN FunctionName Name of computational storage function to query

IN OUT Length Length in bytes of buffer passed for output

OUT Buffer Returns a list of CSENames

6.4.1.3 Description

csQueryCSEList() fills Buffer with a comma separated list of all known CSEs for this query based on FunctionName, where CSEs are identified by CSE names. If Length is sufficient. The function may return zero or more CSEs as a list in Buffer that match the CSF as specified by FunctionName when there are multiple CSE devices on 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 FunctionName is set to NULL, then a list of all CSE devices are returned. Multiple CSEs when returned will be comma separated.

Editor's note: Need mechanism to associate CSE name with a CSx. Query CSEs within a CSx (once CSx is open)

If a valid Buffer pointer is specified of sufficient Length, it is updated with the list of CSE names available and Length updated to actual length of string returned for all functions that match FunctionName. 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 one or more CSEs were available for the list.

Editor's note: how is zero CSEs reported?

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 CSEs 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 CSE device 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 CSE names is able to be parsed and an entry is able to be selected and provided to `csOpenCSE()` to interface with the CSE.

An example source fragment implementation to return all known CSE devices is:

```c
length = 0;
status = csQueryCSEList(NULL, &length, NULL);
if (status != CS_INVALID_LENGTH) {
    // return in error
}
cse_array = malloc(length);
status = csGetCSEList(NULL, &length, &cse_array[0]);
if (status == 0) {
    ...
}
```

### 6.4.2 `csQueryFunctionList()`

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

#### 6.4.2.1 Synopsis

```c
CS_STATUS csQueryFunctionList(char *Path, int *Length, char *Buffer);
```
6.4.2.2 Parameters

IN Path A string that denotes a path to a file, directory that resides on a device or a device path or a CSE or a CSx. The file/directory may indirectly refer to a namespace and partition.

IN OUT Length Length of buffer passed for output

OUT Buffer Returns a list of comma separated function names in Path

6.4.2.3 Description

csQueryFunctionList() fills Buffer with a list of all known functions for the query based on Path if Length is sufficiently specified. The function may return one or more function names as a list in Buffer that match the Path criteria. If Path is set to NULL, then a list of all functions across all CSxes are returned. Multiple functions when returned will be comma separated.

If a valid Buffer pointer is specified of sufficient Length, it is updated with the list of function 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, CSENOENT or CS_ENXIO as defined in 6.3.2.

6.4.2.5 Notes

If the Path input specified a device path, a CSx or a CSE, then the function 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 functions available on the system. 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 function entry, then each entry will be comma separated. This function may still return with success when `Length` is zero.

The returned list of function names may be parsed and a required entry may be selected for further discovery or utilized to interface with a specific CSE.

### 6.4.3 `csGetCSxFromPath()`

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

*Editor’s note: Can a path have multiple CSEs? If so, this is problematic as to which CSEName is returned.*

#### 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 CSx’es when returned will be comma separated.

If a valid `DevName` buffer pointer is specified of sufficient `Length`, 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

\[
\text{CS\_STATUS \hspace{1em} csOpenCSx(char } \text{*DevName, void } \text{*DevContext, CS\_DEV\_HANDLE } \text{*DevHandle);}
\]

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

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

If a valid \text{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 \text{CS\_SUCCESS} if there is no error and the specified CSx was found.

Otherwise, the function returns a status of \text{CS\_INVALID\_ARG}, \text{CS\_ENTITY\_NOT\_ON\_DEVICE}, or \text{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

\[
\text{CS\_STATUS \hspace{1em} csCloseCSx(CS\_DEV\_HANDLE DevHandle);}\]
### 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 `csGetCSEFromCSx()`

This function returns one or more CSE's associated with the specified CSx.

#### 6.5.3.1 Synopsis

```c
CS_STATUS csGetCSEFromCSx(CS_DEV_HANDLE DevHandle,
                          unsigned int *Length, char *CSEName);
```

#### 6.5.3.2 Parameters

- IN DevHandle A handle to the CSx to query
- IN OUT Length Length of buffer passed for output
- OUT CSEName Returns the qualified name to CSE

#### 6.5.3.3 Description

`csGetCSEFromCSx()` queries the CSx device provided by `DevHandle` to return the associated CSEs. The query may return one or more CSEs if the specific CSx contains such. Multiple CSEs when returned will be comma separated.

If a valid `CSEName` pointer is specified of sufficient `Length`, it is updated with the qualified name of one or CSEs 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 `CSEName` 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.

6.5.3.4 Return Value

This function returns CS_SUCCESS if there is no error and a valid CSE was found to be associated with the specified CSx.

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.

6.5.4 csOpenCSE()

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

6.5.4.1 Synopsis

CS_STATUS csOpenCSE(char *CSEName, void *CSEContext, CS_CSE_HANDLE *CSEHandle);

6.5.4.2 Parameters

IN CSEName  A string that denotes the full name of the CSE

IN CSEContext  A user specified context to associate with the CSE for future notifications

OUT CSEHandle  Returns the handle to the CSE device

6.5.4.3 Description

csOpenCSE() opens the CSE and provides a handle for future usages to the user.

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

All input and output parameters are required for this function.

6.5.4.4 Return Value

This function returns CS_SUCCESS if there is no error and the specified CSE 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.
6.5.5 **csCloseCSE()**

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

### 6.5.5.1 Synopsis

```c
CS_STATUS csCloseCSE(CS_CSE_HANDLE CSEHandle);
```

### 6.5.5.2 Parameters

- **IN CSEHandle** Handle to CSE

### 6.5.5.3 Description

A valid `CSEHandle` is required to be provided for this call. The CSE if open is closed and all outstanding requests are terminated.

All input and output parameters are required for this function.

### 6.5.5.4 Return Value

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

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

6.5.6 **csRegisterNotify()**

Register a callback function to be notified based on various computational storage events across all CSx’s and CSEs.

This is an optional function.

### 6.5.6.1 Synopsis

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

*Editor's note: need to determine if this is to the CSE or CSx*

### 6.5.6.2 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
6.5.6.3 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 CSE’s. An invalid input returns an error status.

All input parameters are required for this function.

6.5.6.4 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.

6.5.6.5 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.

6.5.7 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.

6.5.7.1 Synopsis

CS_STATUS csDeregisterNotify(char *DevName, csDevNotificationFn NotifyFn);

6.5.7.2 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

6.5.7.3 Description

csDeregisterNotify() removes a previously provided callback for notifications from one or more CSE’s 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 CSE’s. An invalid input returns an error status.

All input parameters are required for this function.

6.5.7.4 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
CS_STATUS csAllocMem(CS_DEV_HANDLE DevHandle, int Bytes, 
unsigned int MemFlags, CS_MEM_HANDLE *MemHandle, 
CS_MEM_PTR *VAddressPtr);

6.6.1.2 Parameters
IN DevHandle Handle to CSx
IN Bytes Length in bytes of FDM to allocate
IN MemFlags Reserved, shall be zero
OUT MemHandle Pointer to hold the memory handle once allocated
OUT VAddressPtr 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

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

```c
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 `CS_DEVICE_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

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

### 6.7.1.3 Description

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

A valid `Req` structure (see 6.3.4.2.4) is required to initiate the storage IO operation. All fields in the `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;
- 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,
                                ...)
```
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 6.3.4.2.5) is required to initiate the 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 csGetFunction()
Fetches the CSF specified for scheduling compute offload tasks.

6.9.1.1 Synopsis
CS_STATUS csGetFunction(CS_CSE_HANDLE CSEHandle, char *FunctionName, void *Context, CS_FUNCTION_ID *FunctionId);

6.9.1.2 Parameters
IN CSEHandle Handle to CSE
IN FunctionName A pre-specified hardware function name
IN Context A pointer to specify a context to the hardware function loaded
OUT FunctionId A pointer to hold the function id to the function requested if successful

6.9.1.3 Description
csGetFunction() gets the function id to a compute offload function for scheduling compute offload work on the CSE.

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_DEVICE_NOT_AVAILABLE or as defined in 6.3.2.

Notes

Any compute work that needs to be run on the CSE first requires the associated functions to be initialized first. The function name should first be derived from the API csQueryDeviceForComputeList().
See \texttt{csGroupComputeByIds()} for classification of functions by common groups.

### 6.9.2 csStopFunction()

Stops the CSF specified if it is running any tasks.

#### 6.9.2.1 Synopsis

\begin{verbatim}
CS_STATUS csStopFunction(CS_FUNCTION_ID FunctionId);
\end{verbatim}

#### 6.9.2.2 Parameters

- **IN FunctionId**
  
The function id to the CSF to stop.

#### 6.9.2.3 Description

csStopFunction() stops the CSF specified by the function id from further executions. All outstanding requests shall be completed in error on the CSE.

#### 6.9.2.4 Return Value

If there are no errors in stopping the function \texttt{CS_SUCCESS} is returned.

Otherwise, the function returns an error status of \texttt{CS_INVALID_SERVICE} or \texttt{CS_DEVICE_NOT_AVAILABLE} as defined in 6.3.2.

### 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

\begin{verbatim}
CS_STATUS csQueueComputeRequest(CsComputeRequest *Req, void *Context,
                                 csQueueCallbackFn CallbackFn, CS_EVT_HANDLE EventHandle,
                                 u32 *CompValue);
\end{verbatim}

#### 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 6.3.4.2.7) 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

```c
void csHelperSetComputeArg(CsComputeArg *ArgPtr, enum CS_COMPUTE_ARG_TYPE Type, …);
```

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArgPtr</td>
<td>A pointer to the argument in CsComputeRequest to be set.</td>
</tr>
<tr>
<td>Type</td>
<td>The argument type to set. This may be one of the enum values.</td>
</tr>
<tr>
<td>&lt;…&gt;</td>
<td>One or more variables that make up the argument by type.</td>
</tr>
</tbody>
</table>

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

```
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

```c
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 \texttt{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()}.

6.10.4 \texttt{csHelperReconfigureBatchEntry()}

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

6.10.4.1 Synopsis

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

6.10.4.2 Parameters

\begin{itemize}
  \item IN BatchHandle \hspace{1cm} The handle previously allocated for batch requests.
  \item IN Entry \hspace{1cm} The request’s batch entry index that is reconfigured.
  \item IN Req \hspace{1cm} The new batch request entry details.
\end{itemize}

6.10.4.3 Description

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

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.

6.10.5 \texttt{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

- **IN BatchHandle** The handle previously allocated for batch requests that is resized.
- **IN MaxReqs** 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.

6.10.5.3 Description

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

6.10.5.4 Return Value

cSUCCESS 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

- **IN BatchHandle** The handle previously allocated for batch requests.
IN Context A user specified context for the queue request when asynchronous. The parameter is required only if CallbackFn or EventHandle is specified.

IN CallbackFn A callback function if the queue request 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.

### 6.10.6.3 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 verses 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.

```
// 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 one 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 csQueryDeviceForComputeList()

Queries the CSE for its resident CSFs. Functions predefined in the device are returned as an array that will include a count and name.

Editor’s note: May want this for both CSE and CSx

6.12.1.1 Synopsis

CS_STATUS csQueryDeviceForComputeList(CS_DEV_HANDLE DevHandle,
                                int *Size, CsFunctionInfo *FunctionInfo);

6.12.1.2 Parameters

IN DevHandle Handle to CSx

IN OUT Size A pointer to the size of FunctionInfo buffer.

OUT FunctionInfo A pointer to a buffer that is able to hold all the functions resident in the CSE.

6.12.1.3 Description

csQueryDeviceForComputeList() returns a list of fixed CSFs that are resident in the CSE if successful.

If a NULL pointer is specified for FunctionInfo and a valid pointer is provided for Size, the required buffer size is returned back to the user. The user will have to allocate a buffer of the returned size and issue this call again.

It is also possible to provide a large enough buffer and query.

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_OPTION, CS_INVALID_HANDLE, CS_NOT_ENOUGH_MEMORY, or CS_DEVICE_NOT_AVAILABLE as defined in 6.3.2.
6.12.1.5 Notes

The CSE should be queried before any work item is scheduled. This ensures that a valid function is resident. If the user is sure of such a function by name, then this call may be omitted.

The data returned in FunctionInfo is an array of function details. The FunctionId field is able to be used for grouping similar functions together with csGroupComputeByIds() if the user chooses to do so since it aids in optimally scheduling the next function. This need may arise only if there is more than one function with a different name but exact same functionality. The NumUnits field provides the user on the level of parallelism available with this CSF. The Name field identifies the function and is able to be used to load and initialize it when work needs to be scheduled.

The field GroupId provides information about grouping of functions that is enabled through the call csGroupComputeByIds. This is a hint provided by the caller of this function to help hardware associate similar functions. The function scheduler will use this hint while scheduling compute jobs to be optimal across the different units specified by NumUnits for each function across groups. Any function used by name in the group will be scheduled using this hint.

6.12.2 csQueryDeviceProperties()

Queries the CSx for its properties.

6.12.2.1 Synopsis

CS_STATUS csQueryDeviceProperties(CS_DEV_HANDLE DevHandle, int *Length,
                                 CSxProperties *Buffer);

6.12.2.2 Parameters

IN DevHandle Handle to CSx

IN OUT Length Length in bytes of buffer passed for output

IN Buffer A pointer to a buffer that is able to hold all the device properties.

6.12.2.3 Description

csQueryDeviceProperties() fills Buffer with the device properties for the CSx and all its CSEs if Length is sufficiently specified. The function may return an array of one or more CSEs in Buffer.
If a valid `Buffer` pointer of sufficient `Length` is specified, it is updated with the CSx as well as all its CSEs 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`. 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.

If a valid pointer is specified for `Buffer` and a valid pointer is provided for `Length` and the value in `Length` is insufficient for the device properties, then the required buffer size is returned back in `Length`.

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_INVALID_LENGTH`, `CS_NOT_ENOUGH_MEMORY`, or `CS_DEVICE_NOT_AVAILABLE` as defined in 6.3.2.

### 6.12.2.5 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.

### 6.12.3 `csQueryDeviceCapabilities()`

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

#### 6.12.3.1 Synopsis

```c
CS_STATUS csQueryDeviceCapabilities(CS_DEV_HANDLE DevHandle,
                                    CsCapabilities *Caps);
```

#### 6.12.3.2 Parameters

- **IN DevHandle** Handle to CSx
IN Caps A pointer to a buffer that is able to hold all the CSx capabilities

6.12.3.3 Description

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

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_NOTENOUGH_MEMORY, or CS_DEVICE_NOT_AVAILABLE as defined in 6.3.2.

6.12.3.5 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.

6.12.4 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.

6.12.4.1 Synopsis

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

6.12.4.2 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
6.12.4.3 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 FunctionDetails. When used for CSEDetails, the Identifier field refers to the UniqueName field in CSEProperties. Here, if Identifier is set to NULL, the data for the CSE opened will be provided. When used for FunctionDetails, the Identifier refers to the function statistics to be queried.

For a specific CSE's statistics, the Identifier may be set to its unique name available in csQueryDeviceProperties().

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_NO_PERMISSIONS, CS_NOT_ENOUGH_MEMORY, or CS_DEVICE_NOT_AVAILABLE as defined in 6.3.2.

6.12.4.5 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.

6.12.5 csSetDeviceCapability()

Set the CSx's specific capability. A specific capability setting is able to be changed by the requested type.

This is a privileged function.

6.12.5.1 Synopsis

CS_STATUS csSetDeviceCapability(CS_DEV_HANDLE DevHandle,
                                CS_CAP_TYPE Type, CsCapabilityInfo *Details);

6.12.5.2 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

6.12.5.3 Description

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

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_INVALID_HANDLE`, `CS_NOT_ENOUGH_MEMORY`, `CS_DEVICE_NOT_AVAILABLE`, or `CS_NO_PERMISSIONS` as defined in 6.3.2.

6.12.5.5 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.

6.12.6 `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.

6.12.6.1 Synopsis

```c
CS_STATUS csDownload(CS_DEV_HANDLE DevHandle,
                      CsDownloadInfo *ProgramInfo);
```

6.12.6.2 Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN DevHandle</td>
<td>Handle to CSE</td>
</tr>
<tr>
<td>IN Info</td>
<td>A pointer to a buffer that holds the program details to download</td>
</tr>
</tbody>
</table>
6.12.6.3 Description

csdDownload() 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.

6.12.6.4 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 a CSF accept this call. CSx devices that only have fixed functionality fail this call.

6.12.7 csConfig()

Downloads a specified configuration to a CSE or one of its CSFs. The configuration is implementation specific. It is also implementation specific as to how the downloaded configuration is secured.

Editor’s note: How do you specify a specific CSF?

This is a privileged function.

6.12.7.1 Synopsis

CS_STATUS csConfig(CS_CSE_HANDLE CSEHandle, CsConfigInfo *Info);

6.12.7.2 Parameters

IN CSEHandle Handle to CSE

IN Info A pointer to a buffer that holds the configuration details to download

6.12.7.3 Description

csConfig() downloads a configuration to a CSE or one of its CSFs with the suggested configuration details 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_INVALID_ARG`, `CS_UNSUPPORTED`, `CS_INVALID_HANDLE`, `CS_NO_PERMISSIONS`, `CS_LOAD_ERROR`, `CS_NOT_ENOUGH_MEMORY`, `CS_NOT_ENOUGH_MEMORY` or `CS_DEVICE_NOT_AVAILABLE` as defined in 6.3.2.

6.12.7.5 Notes

Only CSxes that contain a CSE that is capable of processing configuration input accept this call.
6.12.8 *csAbortCSE()*
Aborts all outstanding and queued transactions to the CSE.
This is a privileged function.

6.12.8.1 Synopsis
`CS_STATUS csAbortCSE(CS_CSE_HANDLE CSEHandle);`

6.12.8.2 Parameters
IN CSEHandle Handle to CSE

6.12.8.3 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.

6.12.8.4 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 `CS_DEVICE_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.

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

6.12.9.1 Synopsis
`CS_STATUS csResetCSE(CS_CSE_HANDLE CSEHandle);`

6.12.9.2 Parameters
IN CSEHandle Handle to CSE
6.12.9.3 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.

6.12.9.4 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.

6.12.9.5 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

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

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

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

6.14.1.2 Parameters

IN Type
Library support type query

IN OUT Length
Length of buffer passed for output

OUT Buffer
Returns a list of CSEs

6.14.1.3 Description

csQueryLibrarySupport() fills Buffer with a list of all items for query based on Type if Length is sufficiently specified. 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 of sufficient Length, 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 = csQueryDeviceProperties(dev, &props);
if (status != CS_SUCCESS)
    ERROR_OUT("Could not query device properties\n");
if (props.NumFunctions == 0)
    ERROR_OUT("Device does not have any fixed CSx functions\n");
status = csQueryDeviceCapabilities(dev, &caps);
if (status != CS_SUCCESS)
    ERROR_OUT("Could not query device capabilities\n");
if (caps.Functions.Decompression == 0)
    ERROR_OUT("Device does not contain a decompression function\n");
// get the compute engine for this device
Length = sizeof(cseName);
status = csGetCSEFromCSX(dev, &length, &cseName);
status = csOpenCSE(cseName, cseContext, &cse);
// find my CSF from
myFunction = findMyFunction(sInfo, size);
status = csGetFunction(cse, myFunction, NULL, &functId);
if (status != CS_SUCCESS)
    ERROR_OUT("Could not load function\n");
// allocate device and host memory
for (i = 0; i < 2; i++) {
    status = csAllocMem(dev, 4096, 0, &AFDMArray[i], &vaArray[i]);
if (status != CS_SUCCESS)
    ERROR_OUT("AFDM alloc error\n");
}
// allocate request buffer for 3 args
req = calloc(1, sizeof(CsComputeRequest) + (sizeof(CsComputeArg) * 3));
if (!req)
    ERROR_OUT("memory alloc error\n");

Runtime execution may occur in the following way.
// read file content to AFDM via p2p access
if (!pread(hFile, vaArray[0], 4096, 0))
    ERROR_OUT("file read error\n");
// setup work request
req->DevHandle = dev;
req->FunctionId = functId;
req->NumArgs = 3;
argPtr = &req->Args[0];
csHelperSetComputeArg(&argPtr[0], CS_AFDM_TYPE, AFDMArray[0], 0);
csHelperSetComputeArg(&argPtr[1], CS_32BIT_VALUE_TYPE, 4096);
csHelperSetComputeArg(&argPtr[2], CS_AFDM_TYPE, AFDMArray[1], 0);
// do synchronous work 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 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 device and host memory
for (i = 0; i < 2; i++) {
    status = csAllocMem(dev, 4096, 0, &AFDMArray[i], &vaArray[i]);
    if (status != CS_SUCCESS)
        ERROR_OUT("AFDM alloc error\n");
}
// allocate request buffer for 3 args
req = calloc(1, sizeof(CsComputeRequest) + (sizeof(CsComputeArg) * 3));
if (!req)
    ERROR_OUT("memory alloc error\n");
// read file content to AFDM via p2p access
if (!pread(hFile, vaArray[0], 4096, 0))
    ERROR_OUT("file read error\n");
// setup work request
req->DevHandle = dev;
req->FunctionId = functId;
req->NumArgs = 3;
argPtr = &req->Args[0];
csHelperSetComputeArg(&argPtr[0], CS_AFDM_TYPE, AFDMArray[0], 0);
csHelperSetComputeArg(&argPtr[1], CS_32BIT_VALUE_TYPE, 4096);
csHelperSetComputeArg(&argPtr[2], CS_AFDM_TYPE, AFDMArray[1], 0);
// do asynchronous work 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, MyAsyncCbFn, 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 0. 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.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");
// 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.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");
// now queue batch request
status = csQueueBatchRequest(batchHandle, NULL, NULL, evtHandle, NULL);
if (status != CS_SUCCESS)
    ERROR_OUT("Compute exec error\n");
while ((status = csPollEvent(evtHandle, &context)) != CS_SUCCESS) {
    // IO not done; do other work
}

A.4 Applying Hybrid Batch Processing Feature

The following example demonstrates how to use dependency in batch requests to create a hybrid processing model, where 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.

// hybrid batch setup execution
// large storage IO + 8 parallel compute requests + 8 parallel DMA 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->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 = 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.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, 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 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 first need 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_SYSTEMSUPPORTED, &buflen, &buf);
    if (status != CS_SUCCESS)
        ERROR_OUT("Could not query device properties\n");
    // verify if filesystem is supported

    // initialize function
    status = csGetFunction(cseHandle, myFunctionName, NULL, &funcId);
    if (status != CS_SUCCESS)
        ERROR_OUT("Could not init streaming kernel\n");
// setup storage IO for file usage
storReq = calloc(1, sizeof(CsStorageRequest));
if (!storReq)
    ERROR_OUT("memory alloc error\n");

// setup to read 128kb from the start of the file
storReq->Mode = CS_STORAGE_FILE_IO;
storReq->DevHandle = devHandle;
storReq->u.CsFileIo.Type = CS_STORAGE_LOAD_TYPE;
storReq->u.CsFileIo.FileHandle = fd;
storReq->u.CsFileIo.Offset = 0;
storReq->u.CsFileIo.Bytes = 128 * 1024;
storReq->u.CsFileIo.DevMem.MemHandle = inMemHandle;
storReq->u.CsFileIo.DevMem.ByteOffset = 0;
status = csQueueStorageRequest(storReq, storReq, NULL, evtHandle, NULL);
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
}