Computational Storage Architecture and Programming Model

Version 0.8 Revision 0

Abstract: This SNIA document defines recommended behavior for hardware and software that supports Computational Storage.

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
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.
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.
## Revision History

<table>
<thead>
<tr>
<th>Major</th>
<th>Revision</th>
<th>Editor</th>
<th>Date</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>R1</td>
<td>Stephen Bates</td>
<td>02-Feb-2019</td>
<td>Initial conversion of the NVM Programming Model document.</td>
</tr>
<tr>
<td>0.1</td>
<td>R2</td>
<td>Stephen Bates</td>
<td>11-Mar-2019</td>
<td>Added initial draft of Scope section with some initial feedback</td>
</tr>
<tr>
<td>0.1</td>
<td>R3</td>
<td>Stephen Bates</td>
<td>02-April-2019</td>
<td>Added diagram to Scope section and introduced “Computational Function” as a term.</td>
</tr>
<tr>
<td>0.1</td>
<td>R4</td>
<td>Nick Adams</td>
<td>04-Apr-2019</td>
<td>Updates per Morning Session on 03-Apr-2019</td>
</tr>
<tr>
<td>0.1</td>
<td>R5</td>
<td>Scott Shadley</td>
<td>24-Apr-2019</td>
<td>Updates added per F2F meeting in Boston</td>
</tr>
<tr>
<td>0.1</td>
<td>R6</td>
<td>Stephen Bates and Bill Martin</td>
<td>21-Aug-2019</td>
<td>Removed resolved comments, added Theory of Operation, added illustrative diagrams, added placeholder for an illustrative example. Also performed a large amount of editorial clean-up. Added forward and removed taxonomy and abstract (approved by TWG on August 28th 2019).</td>
</tr>
<tr>
<td>0.1</td>
<td>R7</td>
<td>Stephen Bates and Bill Martin</td>
<td>22 Oct 2019</td>
<td>Update foreword section. Add references, abbreviations and definitions.</td>
</tr>
<tr>
<td>0.1</td>
<td>R8</td>
<td>David Slik</td>
<td>12 Nov 2019</td>
<td>Moved fields into tables, updated diagrams.</td>
</tr>
<tr>
<td>0.1</td>
<td>R9</td>
<td>Stephen Bates</td>
<td>14 Nov 2019</td>
<td>Minor editorial changes</td>
</tr>
<tr>
<td>0.1</td>
<td>R10</td>
<td>Bill Martin</td>
<td>20 Nov 2019</td>
<td>Incorporated approved dictionary terms, abbreviations, and reference modifications</td>
</tr>
<tr>
<td>0.1</td>
<td>R11</td>
<td>Bill Martin</td>
<td>11 Dec 2019</td>
<td>Incorporated NetInt proposal for Video Compression CSS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Incorporated ScaleFlux proposal for Database Filter CSS</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Removed previously accepted changes</td>
</tr>
<tr>
<td>0.1</td>
<td>R12</td>
<td>Stephen Bates</td>
<td>15 Dec 2019</td>
<td>Incorporated DellEMC proposal for Hashing and CRC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Removed list of CSSes at start of Section 5.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Remove reference to how to add new CSSes to Section 5.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Update the list of contributors as per the list on the Wiki</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Resolved and removed comments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Added illustrative examples from Eideticom and Arm/NGD Systems in their current form and started merging them correctly into this document.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Incorporated Seagate proposal for large dataset PCSS</td>
</tr>
<tr>
<td>0.1</td>
<td>R13</td>
<td>Stephen Bates</td>
<td>18 Dec 2019</td>
<td>Removed Annex B (Illustrative Examples)</td>
</tr>
<tr>
<td>0.3</td>
<td>R1</td>
<td>Stephen Bates</td>
<td>18 Dec 2019</td>
<td>Update author list</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Slight fix to Philip’s PCSS example</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Add Raymond’s FCSS example</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Minor corrections to document.</td>
</tr>
<tr>
<td>0.4</td>
<td>R1</td>
<td>Bill Martin</td>
<td>8 April 2020</td>
<td>Added PCSS on a Large Multi-Device Dataset</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Added PCSS – Linux Container Example</td>
</tr>
<tr>
<td>0.4</td>
<td>R2</td>
<td>Bill Martin</td>
<td>20 May 2020</td>
<td>Added FCSS – Data Deduplication</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Added PCSS – OPEN CL example</td>
</tr>
<tr>
<td>0.4</td>
<td>R3</td>
<td>Bill Martin</td>
<td>29 July 2020</td>
<td>Updated example in B.4 Data Deduplication Example</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Added Data Compression Illustrative Example</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Added Data Filtration Illustrative Example</td>
</tr>
<tr>
<td>0.4</td>
<td>R4</td>
<td>Bill Martin</td>
<td>5 August 2020</td>
<td>Added Scatter Gather illustrative example</td>
</tr>
<tr>
<td>0.5</td>
<td>R1</td>
<td>Bill Martin</td>
<td>10 Aug 2020</td>
<td>Minor editorial cleanup</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Removed editor’s notes in preparation for public review release</td>
</tr>
<tr>
<td>0.5</td>
<td>R2</td>
<td>Bill Martin</td>
<td>12 Aug 2020</td>
<td>Modifications created during TWG call on August 12. There are some partial sentences as the discussion was ended due to the end of the call.</td>
</tr>
<tr>
<td>0.5</td>
<td>R3</td>
<td>Bill Martin</td>
<td>8 September</td>
<td>Includes editor’s comments removed from public review version</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Includes comments received to date (Intel (Kim) and HPe (Chris))</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Changes for the terminology of CSS, FCSS, PCSS, and CSP</td>
</tr>
</tbody>
</table>

---

Computational Storage Architecture and Programming Model
Working Draft Version 0.8 rev 0
<table>
<thead>
<tr>
<th>Number</th>
<th>Version</th>
<th>Author</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>R4</td>
<td>Bill Martin</td>
<td>30 September</td>
<td>Renaming of components (e.g., addition of Computational Storage Engine, Computational Storage Service becomes Computational Storage Function)</td>
</tr>
<tr>
<td>0.5</td>
<td>R5</td>
<td>Bill Martin</td>
<td>9 November</td>
<td>Changes from meetings</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Modified figures as requested</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Added memory definitions</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Removed resolved comments</td>
</tr>
<tr>
<td>0.5</td>
<td>R6</td>
<td>Bill Martin</td>
<td>10 Dec 2020</td>
<td>Removed change tracking R4 to R5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fixed figure 1.1 to include CFM</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Added brief descriptions of CSFM, FDM, AFDM in 4.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Moved example discover request and response to an annex</td>
</tr>
<tr>
<td>0.5</td>
<td>R7</td>
<td>Scott Shadley</td>
<td>16 Dec 2020</td>
<td>Comments from TWG meeting</td>
</tr>
<tr>
<td>0.5</td>
<td>R8</td>
<td>Scott Shadley</td>
<td>1 Jan 2021</td>
<td>Comments from TWG meeting</td>
</tr>
<tr>
<td>0.5</td>
<td>R9</td>
<td>Bill Martin</td>
<td>28 Jan 2021</td>
<td>Updated direct/indirect model</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Moved discovery/configuration request/response to annex</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Other editorial changes from Jan 27, 2021 meeting</td>
</tr>
<tr>
<td>0.5</td>
<td>R10</td>
<td>Bill Martin</td>
<td>28 Jan 2021</td>
<td>Removed redline and completed comments</td>
</tr>
<tr>
<td>0.5</td>
<td>R11</td>
<td>Bill Martin</td>
<td>10 Feb 2021</td>
<td>Resolved a number of comments from TWG meetings</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>See individual comments with resolutions</td>
</tr>
<tr>
<td>0.5</td>
<td>R12</td>
<td>Bill Martin</td>
<td>11 Feb 2021</td>
<td>Removed redline and completed comments</td>
</tr>
<tr>
<td>0.5</td>
<td>R13</td>
<td>Bill Martin</td>
<td>16 Feb 2021</td>
<td>Modifications from TWG meeting</td>
</tr>
<tr>
<td>0.5</td>
<td>R14</td>
<td>Bill Martin</td>
<td>23 Mar 2021</td>
<td>Updated Large Data Set CSFs</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Updating Appendix B, sect 5 (Compression) and 6 (data filtering) illustrative examples to align with the CSx/CSE/CSF terminology, (change)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Removed all resolved comments</td>
</tr>
<tr>
<td>0.5</td>
<td>R15</td>
<td>Bill Martin</td>
<td>23 April 2021</td>
<td>Updates from work group meetings with change bars</td>
</tr>
<tr>
<td>0.5</td>
<td>R16</td>
<td>Bill Martin</td>
<td>27 April 2021</td>
<td>Changes to 4.2 and 4.3 as agreed to in TWG meeting</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Incorporated architecture model changes agreed in TWG meeting</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Incorporated changes to section 5 as agreed in TWG meeting</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Incorporate changes for Appendix B from 4/14/2021</td>
</tr>
<tr>
<td>0.5</td>
<td>R17</td>
<td>Bill Martin</td>
<td>27 April 2021</td>
<td>Accepted changes and removed resolved comments</td>
</tr>
<tr>
<td>0.5</td>
<td>R18</td>
<td>Bill Martin</td>
<td>27 May 2021</td>
<td>Added Configuration flow chart per proposal</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Updated Open CL CSS per proposal</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Updated figures in Usage section</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Updated Usage definition to make consistent</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Moved open comments to “editor’s notes”</td>
</tr>
<tr>
<td>0.5</td>
<td>R19</td>
<td>Bill Martin</td>
<td>3 June 2021</td>
<td>Incorporated modifications to B.2</td>
</tr>
<tr>
<td>0.8</td>
<td>R0</td>
<td>Arnold Jones</td>
<td>9 June 2021</td>
<td>Formatted as a Working Draft for release outside the TWG per TWG approval ballot.</td>
</tr>
</tbody>
</table>
Table of Contents

FOREWORD...........................................................................................................13
  1.1 SCOPE ............................................................................................................. 13

2 REFERENCES..........................................................................................................15

3 DEFINITIONS, ABBREVIATIONS, AND CONVENTIONS....................................16
  3.1 DEFINITIONS .................................................................................................. 16
  3.2 KEYWORDS .................................................................................................... 17
  3.3 ABBREVIATIONS ............................................................................................ 18
  3.4 CONVENTIONS ............................................................................................... 18

4 THEORY OF OPERATION ......................................................................................20
  4.1 OVERVIEW ..................................................................................................... 20
  4.2 DISCOVERY .................................................................................................... 23
  4.3 CONFIGURATION ............................................................................................ 24
  4.4 CSF MONITORING .......................................................................................... 26
  4.5 SECURITY ........................................................................................................ 26
  4.6 CSF USAGE ..................................................................................................... 26

5 EXAMPLE COMPUTATIONAL STORAGE FUNCTIONS .......................................30
  5.1 COMPRESSION CSF ...................................................................................... 30
  5.2 DATABASE FILTER CSF ................................................................................ 30
  5.3 ENCRYPTION CSF ........................................................................................ 30
  5.4 ERASURE CODING CSF ................................................................................ 30
  5.5 RegEx CSF ....................................................................................................... 31
  5.6 SCATTER-GATHER CSF .................................................................................. 31
  5.7 PIPELINE CSF ................................................................................................ 31
  5.8 VIDEO COMPRESSION CSF .......................................................................... 31
  5.9 HASH/CRC CSF .............................................................................................. 32
B.2.4 Usage .................................................................................................................................................. 46

Once the configuration process has completed and both CSEE1 and CSEE2 are activated and the CSF1 is ready to process data, the usage process steps (as shown in Figure B.2.8) can occur: .................................................................................................................................................. 46

1) The CSF1 is told by the host to execute the function on data; .......................................................... 46

2) Input data is pulled from device storage into device memory, in the allocated function data memory (AFDM) space of the function data memory (FDM); .......................................................... 46

3) The data is operated on by the CSF1 and is then stored as output data in the device storage at the location specified by the function; and .......................................................... 46

4) The host can then access both the original input data and the new output data as required to complete the process steps. .................................................................................................................................................. 46

B.2.5 Example of Application Deployment .......................................................................................................................... 47

B.3 Data Deduplication CSF ............................................................................................................................................. 49

B.3.1 Overview ........................................................................................................................................................... 49

B.3.2 Theory of Operation ........................................................................................................................................... 51

B.3.3 Discovery ............................................................................................................................................................ 52

B.3.4 Configuration ....................................................................................................................................................... 52

B.3.5 Operation ........................................................................................................................................................... 54

B.3.6 Monitoring .......................................................................................................................................................... 54

B.3.7 CSF Data Deduplication Example ............................................................................................................................. 54

B.4 A Computational Storage Function on a NVM Express and OpenCL Based Computational Storage Drive Illustrative Example .................................................................................................................. 60

B.4.1 CSEE Example ................................................................................................................................................... 60

B.4.2 Computational Storage Drive ............................................................................................................................... 61

B.4.3 Theory of Operation ........................................................................................................................................... 62

B.4.4 Discovery ............................................................................................................................................................ 62

B.4.4.1 NVMe Function Discovery ................................................................................................................................... 62

B.4.4.2 OpenCL CSP Function Discovery ............................................................................................................................. 63
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.4.5</td>
<td><strong>CONFIGURATION – EXPLICIT MODE</strong></td>
<td>64</td>
</tr>
<tr>
<td>B.4.6</td>
<td><strong>USAGE – STORAGE DIRECT</strong></td>
<td>64</td>
</tr>
<tr>
<td>B.4.7</td>
<td><strong>USAGE – EXPLICIT MODE COMPUTATIONAL STORAGE</strong></td>
<td>66</td>
</tr>
<tr>
<td>B.5</td>
<td><strong>DATA COMPRESSION CSF EXAMPLE</strong></td>
<td>68</td>
</tr>
<tr>
<td>B.5.1</td>
<td><strong>OVERVIEW</strong></td>
<td>68</td>
</tr>
<tr>
<td>B.5.2</td>
<td><strong>THEORY OF OPERATION</strong></td>
<td>68</td>
</tr>
<tr>
<td>B.5.3</td>
<td><strong>DISCOVERY</strong></td>
<td>68</td>
</tr>
<tr>
<td>B.5.4</td>
<td><strong>CONFIGURATION</strong></td>
<td>69</td>
</tr>
<tr>
<td>B.5.5</td>
<td><strong>MONITORING</strong></td>
<td>69</td>
</tr>
<tr>
<td>B.6</td>
<td><strong>DATA FILTER CSF EXAMPLE</strong></td>
<td>70</td>
</tr>
<tr>
<td>B.6.1</td>
<td><strong>OVERVIEW</strong></td>
<td>70</td>
</tr>
<tr>
<td>B.6.2</td>
<td><strong>THEORY OF OPERATION</strong></td>
<td>70</td>
</tr>
<tr>
<td>B.6.3</td>
<td><strong>DISCOVERY</strong></td>
<td>71</td>
</tr>
<tr>
<td>B.6.4</td>
<td><strong>CONFIGURATION</strong></td>
<td>71</td>
</tr>
<tr>
<td>B.6.5</td>
<td><strong>OPERATION</strong></td>
<td>71</td>
</tr>
<tr>
<td>B.6.6</td>
<td><strong>MONITORING</strong></td>
<td>72</td>
</tr>
<tr>
<td>B.7</td>
<td><strong>SCATTER GATHER CSF</strong></td>
<td>73</td>
</tr>
<tr>
<td>B.7.1</td>
<td><strong>OVERVIEW</strong></td>
<td>73</td>
</tr>
<tr>
<td>B.7.1.1</td>
<td><strong>REPRESENTATION OF A DATA MOVEMENT REQUEST (DMR)</strong></td>
<td>73</td>
</tr>
<tr>
<td>B.7.1.2</td>
<td><strong>ENDPOINTS</strong></td>
<td>74</td>
</tr>
<tr>
<td>B.7.1.3</td>
<td><strong>FLOWS</strong></td>
<td>75</td>
</tr>
<tr>
<td>B.7.1.4</td>
<td><strong>EXAMPLE DMRs</strong></td>
<td>76</td>
</tr>
<tr>
<td>B.7.2</td>
<td><strong>SCATTER-GATHER CSS THEORY OF OPERATION</strong></td>
<td>78</td>
</tr>
<tr>
<td>B.7.2.1</td>
<td><strong>DISCOVERY</strong></td>
<td>78</td>
</tr>
<tr>
<td>B.7.2.2</td>
<td><strong>CONFIGURATION</strong></td>
<td>78</td>
</tr>
</tbody>
</table>
B.7.2.3 OPERATION .......................................................... 79
B.7.2.4 MONITORING .......................................................... 79
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>An Architectural view of Computational Storage</td>
<td>21</td>
</tr>
<tr>
<td>4.2</td>
<td>Discovery Flowchart</td>
<td>24</td>
</tr>
<tr>
<td>4.3</td>
<td>CSF Discovery and Configuration Flowchart</td>
<td>25</td>
</tr>
<tr>
<td>4.4</td>
<td>Direct Usage Model</td>
<td>27</td>
</tr>
<tr>
<td>4.5</td>
<td>Indirect Usage Interactions</td>
<td>28</td>
</tr>
<tr>
<td>B.1.1</td>
<td>Ceph, a scale-out storage system</td>
<td>38</td>
</tr>
<tr>
<td>B.1.2</td>
<td>Ceph CSA</td>
<td>39</td>
</tr>
<tr>
<td>B.1.3</td>
<td>Ceph CSDs</td>
<td>39</td>
</tr>
<tr>
<td>B.1.4</td>
<td>Skyhook Example 1</td>
<td>41</td>
</tr>
<tr>
<td>B.1.5</td>
<td>Skyhook Command Execution</td>
<td>42</td>
</tr>
<tr>
<td>B.2.6</td>
<td>CSx discovery process</td>
<td>44</td>
</tr>
<tr>
<td>B.2.7</td>
<td>CSx Configuration process</td>
<td>45</td>
</tr>
<tr>
<td>B.2.8</td>
<td>CSx usage</td>
<td>46</td>
</tr>
<tr>
<td>B.3.1</td>
<td>Simplified Data Deduplication Process</td>
<td>49</td>
</tr>
<tr>
<td>B.3.2</td>
<td>Post Process Data Deduplication</td>
<td>50</td>
</tr>
<tr>
<td>B.3.3</td>
<td>Inline Data Deduplication</td>
<td>51</td>
</tr>
<tr>
<td>B.3.4</td>
<td>Configuration Parameters</td>
<td>53</td>
</tr>
<tr>
<td>B.3.5</td>
<td>Volume Creation</td>
<td>55</td>
</tr>
<tr>
<td>B.3.6</td>
<td>Enable Deduplication</td>
<td>55</td>
</tr>
<tr>
<td>B.3.7</td>
<td>Retrieve Status</td>
<td>56</td>
</tr>
<tr>
<td>B.3.8</td>
<td>Disable Deduplication Scheduling</td>
<td>56</td>
</tr>
<tr>
<td>B.3.9</td>
<td>Copy files</td>
<td>57</td>
</tr>
<tr>
<td>B.3.10</td>
<td>Verify space utilized</td>
<td>57</td>
</tr>
<tr>
<td>B.3.11</td>
<td>Schedule deduplication</td>
<td>58</td>
</tr>
<tr>
<td>B.3.12</td>
<td>Monitor Progress</td>
<td>58</td>
</tr>
<tr>
<td>B.3.13</td>
<td>verify space savings</td>
<td>59</td>
</tr>
<tr>
<td>B.4.1</td>
<td>CS architectural diagram adapted for this illustrative example</td>
<td>61</td>
</tr>
<tr>
<td>B.6.1</td>
<td>An example to illustrate the function of data filter FCSS</td>
<td>70</td>
</tr>
<tr>
<td>B.7.1</td>
<td>Top-level DMR structure</td>
<td>74</td>
</tr>
<tr>
<td>B.7.2</td>
<td>Example endpoint definition structure for an NVMe endpoint</td>
<td>74</td>
</tr>
<tr>
<td>B.7.3</td>
<td>Flow definition structure</td>
<td>75</td>
</tr>
<tr>
<td>B.7.4</td>
<td>Scatter DMR</td>
<td>76</td>
</tr>
<tr>
<td>B.7.5</td>
<td>Example Scatter DMR</td>
<td>77</td>
</tr>
<tr>
<td>B.7.6</td>
<td>Pipeline DMR</td>
<td>77</td>
</tr>
<tr>
<td>B.7.7</td>
<td>Example Pipeline DMR</td>
<td>78</td>
</tr>
</tbody>
</table>
FOREWORD

The SNIA Computational Storage Technical Working Group was formed to establish architectures and software computation in its many forms to be more tightly coupled with storage, at both the system and drive level. An architecture model and a programming model are necessary to allow vendor-neutral, interoperable implementations of this industry architecture.

This SNIA specification outlines the architectural models that are defined to be Computational Storage, and a programming model for those architectures and functions. As this specification is developed, requirements in interface standards and specific APIs may be proposed as separate documents and developed in the appropriate organizations.

1.1 Scope

This specification focuses on defining the capabilities and actions that are able to be implemented across the interface between Computational Storage devices (CSxes) (e.g. Computational Storage Processors, Computational Storage Drives and Computational Storage Arrays) and either Host Agents or other CSxes.

The actions mentioned above are associated with several aspects of a CSx:

- **Management.** Actions that allow Host Agent(s), based on security policies, to perform:
  - **Discovery.** Mechanisms to identify and determine the capabilities and Computational Storage Resources (CSR) and Computational Storage Functions (CSF).
  - **Configuration.** Programming parameters for initialization, operation, and/or resource allocation
  - **Monitoring.** Reporting mechanisms for events and status

- **Security.** Actions that allow Host Agent(s) and/or CSx to perform:
  - **Authentication.** Host agent to CSx and CSx to host agent
  - **Authorization.** Mechanisms for secure data access and permissions control
  - **Encryption.** Mechanisms to perform computation on encrypted data that was not encrypted by the CSx. Mechanisms that exchange information necessary for the CSx to encrypt/decrypt data.
  - **Auditing.** Mechanisms that allow for generating and retrieving of a secure log

- **Usage.** Mechanisms for the CSx to store and retrieve data. Allows a Host Agent or CSx to offload Computational Storage tasks to a CSx, including providing the target CSx with information about data locality both local to the CSx or resident on one or more non-local locations.

This specification makes no assumptions about the physical nature of the interface between the Host Agent and CSx(s). This specification and the actions associated with it will be implemented across a range of different physical interfaces. This specification also makes no assumptions about the storage protocols used by Host Agents and CSxes.

The following storage protocols between the Host Agent and the CSx may be supported:
• **Logical Block Address.** Data is grouped into fixed-size logical units and operations are atomic at that unit size. Data is indexed via a numerical index into the Logical Block Address.
• **Key-Value.** Data is not fixed-size and is indexed by a key.
• **Persistent Memory.** Byte addressable non-volatile memory.

This specification defines actions for passing data through multiple Computational Storage Functions that may or may not reside on a single CSx. Additionally, it defines actions for requesting multiple Computational Storage Functions to perform a set of tasks.
2 References

The following referenced documents are indispensable for the application of this document.

For references available from ANSI, contact ANSI Customer Service Department at (212) 642-4900 (phone), (212) 302-1286 (fax) or via the World Wide Web at http://www.ansi.org.

NVMe 1.4    NVM Express Revision 1.4,
Approved standard, available from http://nvmexpress.org
3 Definitions, abbreviations, and conventions
For the purposes of this document, the following definitions and abbreviations apply.

3.1 Definitions

3.1.1 Allocated Function Data Memory
Function Data Memory (FDM) that is allocated for a particular instance of an API

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

3.1.3 Computational Storage Array (CSA)
Storage array that contains one or more CSEs.

3.1.4 Computational Storage Device (CSx)
Computational Storage Drive, Computational Storage Processor, or Computational Storage Array.

3.1.5 Computational Storage Drive (CSD)
storage element that contains one or more CSEs and persistent data storage.

3.1.6 Computational Storage Engine (CSE)
Component that is able to execute one or more CSFs

note 1 to entry Examples are: CPU, FPGA.

3.1.7 Computational Storage Function (CSF)
a set of specific operations that may be configured and executed by a CSE.

Note 1 to entry Examples are: compression, RAID, erasure coding, regular expression, encryption.

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

3.1.9 Computational Storage Processor (CSP)
component that contains one or more CSEs for an associated storage system without providing persistent data storage
3.1.10 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 (e.g., CSP, CPU, memory, FPGA resources)

3.1.11 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

Editor’s note: Delete if not used

3.1.12 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

3.1.13 Key Value
Storage that stores and retrieves user data based on a key that is associated with that data

3.1.14 Object Store
storage that accesses data as objects and provides services for storing, searching and returning
that data based on content of the storage device

Editor’s note: Delete if not used

3.2 Keywords
In the remainder of the specification, the following keywords are used to indicate text related to
compliance:

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

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

3.2.3 may not
keywords that indicate flexibility of choice with no implied preference; “may not” is equivalent to
“may or may not”
3.2.4 need not
keywords indicating a feature that is not required to be implemented; “need not” is equivalent to “is not required to”

3.2.5 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

3.2.6 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

3.2.7 should
a keyword indicating flexibility of choice with a strongly preferred alternative

3.3 Abbreviations
AFDM Allocated Function Data Memory
CSA Computational Storage Array
CSD Computational Storage Drive
CSE Computational Storage Engine
CSF Computational Storage Function
CSP Computational Storage Processor
CSR Computational Storage Resources
CSx Computational Storage devices
FDM Function Data Memory
NVM Non-Volatile Memory
PM Persistent Memory
SSD Solid State Disk

3.4 Conventions
Representation of modes in figures
Modes are represented by red, wavy lines in figures, as shown below:

The wavy lines have labels identifying the mode name (which in turn, identifies a section of the specification).
4 Theory of Operation

4.1 Overview

This section describes the theory of operations for Computational Storage Devices (CSxes), Computational Storage Resources (CSRs), Computational Storage Engines (CSEs), Computational Storage Engine Environments (CSEEs), and Computational Storage Functions (CSFs).

Computational Storage 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.
Computational Storage Architecture consists of the following components:

- A Storage Controller for CSD or an Array Controller for CSA;
- Computational Storage Resources (CSR) which contain:
  - A Resource Repository where the following may be stored:
    - Computational Storage Functions (CSFs); and
    - Computational Storage Engine Environments (CSEEs);
  - Function Data Memory (FDM) which may be partitioned into Allocated Function Data Memory (AFDM); and
  - One or more Computational Storage Engines (CSEs);
- A Storage Controller for CSD or an Array Controller for CSA;

**Figure 4.1– An Architectural view of Computational Storage**

An illustrative example of Computational Storage devices (CSxes) is shown in Figure 4.1. A CSx consists of the following components:
• Device Memory; and
• Device Storage for CSD and CSA.

Computational Storage Resources (CSRs) are the resources available in a CSx necessary for that CSx to store and execute a CSF.

A Computational Storage Engine (CSE) is a CSR that is able to be programmed to provide one or more specific operation(s). A CSE is required to have a CSEE activated to be able to have a CSF activated. A CSE has FDM associated with it. A CSE is able to have one or more CSEEs and one or more CSFs activated at the time of manufacture that are usable by the host via management and I/O interfaces, or it is able to be have one or more CSEEs and one or more CSFs downloaded by the host and activated. A CSE may have CSFs that have been programmed at the time of manufacture that are not changeable (i.e., not stored in the Resource Repository) (e.g., compression, RAID, erasure coding, regular expression, encryption). CSFs that are stored in the Resource Repository may be activated in a CSEE in a CSE.

A Computational Storage Engine Environment (CSEE) is an operating environment space for the CSE. A CSEE may be pre-installed or downloaded by the host. A downloaded CSEE or pre-installed CSEE is required to be activated for use. A CSEE may support the ability to have additional CSEEs activated within it.

A Computational Storage Function (CSF) is a set of specific operations that may be configured and executed by a CSE. The CSF performs only the defined operations (e.g., a specific eBPF program or compression) that are reported by the CSx (i.e., the underlying operation is not changeable).

Function Data Memory (FDM) is device memory that is available for CSFs to use for data that is used or generated as part of the operation of the CSF.

Allocated Function Data Memory (AFDM) is a portion of FDM that is allocated for one or more specific instances of a CSF operation.

The Resource Repository is a region located within the CSx that contains zero or more images for CSFs and CSEEs that are available for activation. These CSFs and CSEEs are required to be activated in the CSE in order to be utilized. The Repository may be in memory or in storage.

A Computational Storage Processor (CSP) is a component that is able to execute one or more CSFs for an associated storage system without providing persistent data storage. The CSP contains CSRs and Device Memory, The mechanism by which the CSP is associated with the storage system is implementation specific.

A Computational Storage Drive (CSD) is a component that is able to execute one or more CSFs and provides persistent data storage. The CSD contains a Storage Controller, CSR, Device Memory, and persistent data storage.

A CSD may continue to function as a standard Storage Drive, with existing host interfaces and drive functions. As such, the system is able to have a storage controller with associated storage
memory, along with storage addressable by the host through standard management and I/O interfaces.

A Computational Storage Array (CSA) is a storage array that is able to execute one or more CSFs. As a storage array, a CSA contains control software, which provides virtualization to storage services, storage devices, and CSRs for the purpose of aggregating, hiding complexity or adding new capabilities to lower level storage resources. The CSRs in the CSA may be centrally located or distributed across CSDs/CSPs within the array.

4.2 Discovery

4.2.1 CSx Discovery Overview

Discovery of CSxes is fabric dependent and is outside of the scope of this architecture.

4.2.2 CSR Discovery Overview

Once a CSx is discovered, to utilize Computational Storage Resources (CSRs), the characteristics of that CSx needs to be discovered. This involves a CSR discovery process for each discovered CSx. The CSR discovery process discovers all resources available including CSEs, CSEEis, CSFs, and FDM.

Discovery of a CSE includes information of any activated CSEEis and any activated CSFs in those CSEEis.

CSEEis in the Resource Repository may be discovered and information about any CSFs pre-activated in those CSEEis is returned. CSEEis in the Resource Repository are required to be activated in order to be used.

CSFs in the Resource Repository may be discovered. CSFs in the Resource Repository are required to be activated in order to be used.

The specifics of a CSR discovery process are defined by an API specification.
Figure 4.2 shows an example flowchart of discovery of resources and functions on a CSx.
4.3 Configuration

4.3.1 CSE Configuration Overview
A CSE may be configured to prepare it for use. One aspect of CSE configuration is activation of one or more CSEEs. A CSEE may be activated in the CSE at time of manufacture and therefore not be required to be activated as part of configuration. The specifics of a CSE configuration process is defined by an API specification.

4.3.2 CSEE Configuration Overview
A CSEE may be configured to prepare it for use. A CSEE is required to be activated in order to be used by a CSE. One aspect of CSEE configuration is activation of one or more CSFs. A CSF may be pre-activated in the CSEE and therefore not be required to be activated as part of configuration. The specifics of a CSEE configuration process is defined by an API specification.
4.3.3 CSF Configuration Overview

A Computational Storage Function may be configured to prepare it for use. A CSF is required to be activated in order to be used by a CSEE. The specifics of a CSF configuration process is defined by an API specification.

This process may be done once for the CSF, prior to any specific invocation of the CSF, or as parameters associated with the invocation of a CSF.

4.3.4 Memory Configuration

4.3.5 Resource reservations

4.3.6 CSF Discovery and Configuration Example

Figure 4.3 shows an example flowchart of discovery and configuration of a CSF. This example assumes that each of the actions can be completed and that there are no errors. This is only one example of how configuration is able to be completed.

![Figure 4.3 – CSF Discovery and Configuration Flowchart](image-url)
The flow of the discovery and configuration process is a number of steps to determine what is already activated in the CSX. For a CSF or CSEE that is not already activated there is a discovery if the desired CSF or CSEE exists in the Resource Repository. If a desired CSF or CSEE is not in the Resource Repository then it has to be downloaded to the Resource Repository.

For a desired CSEE that is not activated, that CSEE is required to be activated in a CSE. After the desired CSEE is activated and the desired CSF is available in the Resource Repository, that CSF is activated in the CSEE.

For a desired CSF that is activated in a CSEE, that CSF is configured, if there are static configurations that are required for all executions of that CSF. Once the CSF is configured it is available for an application to execute.

4.4 CSF Monitoring

This section will be filled in at a later date.

4.5 Security

This section will be filled in as the security model is completed.

4.6 CSF Usage

4.6.1 CSF Usage Overview

Once configured, a host may use the CSF with:

a) a direct usage model; or
b) an indirect usage model.

In the direct usage model, the host sends a computation request that specifies a CSF to perform on data in the FDM. The data movement between host or storage and the FDM may be done outside of the operation of the CSF.

In the indirect usage model, the host sends a storage request to the Storage Controller. A CSF is performed on the data associated with a storage request based on:

a) parameters in the storage request;
b) the data locality; or
c) the data characteristics (e.g., size).

For the indirect usage model that operates on data based on locality or characteristics, the Storage Controller is configured to associate a CSF with data locality or data characteristics prior to sending a storage request.
This is accomplished through the usage processes illustrated in Figure 4.4 and Figure 4.5.

Figure 4.4 assumes that data on which computation is to be performed is placed in the FDM, prior to the request to the CSE, through some process that is not shown in this figure. The result data, if any, is returned to the host through some process that is not shown in this figure. The steps shown in Figure 4.4 for a direct usage model are:

1. The host sends a command to invoke the CSF;
2. The CSE performs the requested computation on data that is in FDM and places the result, if any, into FDM; and
3. The CSE returns a response to the host.
Figure 4.5 assumes a read operation with computation on the data that is being read. The steps shown in Figure 4.5 to perform an indirect computation through the Storage Controller are:

1. The host sends a storage request to a Storage Controller where:
   a. that storage request is associated with a target CSF; and
   b. the storage controller determines what CSF is associated with the storage request;
2. The Storage Controller moves data from storage into the FDM;
3. The Storage Controller instructs the CSE to perform the indicated computation on the data in the FDM;
4. The CSE performs the computation on the data and places the result, if any, into the FDM; and
5. The Storage Controller returns the computation results, if any, from the FDM to the host.

4.6.2 CSF Command

A CSF command is specific to the type of CSF (e.g., for a compression CSF, a command may instruct the CSF to read from a given location in system memory, compress the data, and store the resulting data to a specified location in a storage device).
4.6.3 CSF Command Response

A CSF command response is specific to the corresponding command.
5 Example Computational Storage Functions

This section describes example Computational Storage Functions (CSFs) (see 4.1).

See section (See 4.2 and 4.3) for information about CSF discovery and configuration.

5.1 Compression CSF

A compression CSF reads data from a source location, compresses or decompresses the data, and writes the result to a destination location.

CSF configuration specifies the compression algorithm and associated parameters.

CSF command specifies the source address and length and the destination address and maximum lengths.

5.2 Database Filter CSF

A database filter CSF reads data from source location(s), performs a database projection (column selection) and filter (row selection) on the data according to projection and filter conditions, and writes the result(s) to destination location(s).

CSF configuration specifies the database format, table schema, selection and filter conditions, and associated parameters.

CSF command specifies the source address and length, and the destination addresses and lengths.

5.3 Encryption CSF

An encryption CSF reads data from a source location, encrypts or decrypts the data, and writes the result to a destination location.

CSF configuration specifies the encryption algorithm, keying information, and associated parameters.

CSF command specifies the source address and length and the destination address and length.

5.4 Erasure Coding CSF

An erasure coding CSF reads data from source location(s), performs a EC encode or decode on the data, and writes the result(s) to destination location(s).

CSF configuration specifies the EC algorithm and associated parameters.
CSF command specifies the source address and length and the destination addresses and lengths.

### 5.5 RegEx CSF

A regex CSF reads data from source location(s), performs a regular expression pattern matching or transformation on the data, and writes the result(s) to the destination location.

CSF configuration specifies the RegEx string(s) and associated parameters.

CSF command specifies the source address and length and the destination address and length.

### 5.6 Scatter-Gather CSF

A Scatter-Gather CSF reads data from set of source location(s) and writes the data to a set of destination location(s).

CSF configuration does not have any parameters.

CSF command specifies the source addresses and lengths and the destination addresses and lengths.

### 5.7 Pipeline CSF

A Pipeline CSF performs a series of operations on data according to a data flow specification, allowing different CSF commands to be combined together in a standardized way.

CSF configuration does not have any parameters.

CSF command specifies a collection of commands, their order and dependencies, and calculations defining the relationships of the addresses between commands.

### 5.8 Video Compression CSF

A video compression CSF reads data from a source location, compresses or decompresses the video, and writes the result to a destination location. In order to accommodate multiple parallel compressions, the video compression CSF may support a single compression stream or multiple compression streams.

CSF configuration specifies the stream, compression algorithm and associated parameters.

CSF command specifies the stream, source address and length and the destination address and maximum lengths.
5.9 Hash/CRC CSF

A hash/CRC CSF reads data from a source location, calculates a hash or CRC value based on the source data, and writes the result to a destination location.

CSF configuration specifies the hashing/CRC algorithm and associated parameters.

CSF command specifies the source address and length and the destination address.

As an optional feature this CSF may calculate the hash/CRC value based on the source data and compare the hash/CRC result to a pre-calculated value supplied by the initiator. The CSS will notify the initiator whether the calculated value matches the supplied value.

5.10 Data Deduplication CSF

A data deduplication CSF reads data from source location(s), performs deduplication or duplication on the data, and writes the result(s) to the destination location(s). CSF configuration specifies the data deduplication algorithm and associated parameters.

CSF command specifies the source address and length and the destination address and maximum lengths.

5.11 Large Data Set CSFs

This example is for a large data set wherein the data is sharded as objects across a plurality of computational storage devices (CSxes) and these objects are further tagged as belonging to a named object class. The object class being defined as a set of methods that act on those named objects. The object class is the CSF. The object class subsystem is the CSE. There are CSEs defined and configured in each of the CSxes. The FDM, and AFDM is pulled from system memory CSF configuration includes the object class methods.

The CSF command specifies the objects names to be acted upon, the object class method to enact and other parameters for the object class model.
6 Example Computational Storage Execution Environment
This section describes example Computational Storage Execution Environments (CSEEs) (see 4.1).

See section (see 4.2 and 4.3) for information about CSEE discovery, configuration, and activation.

6.1 Operating System CSEE
An Operating System CSEE provides a specific operating system environment (e.g., Linux). The Operating System CSEE may contain one or more activated CSFs and may support the activation of one or more downloaded CSFs.

6.2 Container Platform CSEE
A Container Platform CSEE provides an environment to host one or more Container CSEEs. This type of CSEE must be configured with a Container CSEE in order to provide CSFs.

6.3 Container CSEE
A Container CSEE provides a container environment. The Container CSEE may contain one or more activated CSFs and may support the activation of one or more downloaded CSFs.

6.4 eBPF CSEE
A Berkeley Packet Filter (eBPF) CSEE provides an environment for running eBPF programs. The eBPF CSEE may contain one or more activated eBPF CSFs and supports the activation of one or more downloaded eBPF CSFs.

6.5 FPGA Bitstream CSEE
A FPGA Bitstream CSEE provides an environment for an FPGA device. The FPGA Bitstream CSEE may contain one or more activated CSFs and may support the activation of one or more downloaded CSFs.
Annex A. (Informative) Example CSR discovery request/response

A.1 Overview
This annex provides an example of a CSR discovery request/response that may be used to understand the types of information that would be requested and returned for the discovery of the resources available on a given CSx. This is an example only and is not normative.

A.1.1 CSR Discovery Request
An example CSR discovery request may contain the following information:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSx Identifier</td>
<td>Unique identifier of the CSx target</td>
</tr>
<tr>
<td>Host Identifier</td>
<td>Unique identifier of the Host</td>
</tr>
<tr>
<td>CSR filter</td>
<td>Restricts discovery to only CSRs that match the specified filter (e.g., all CSRs, specific types of CSRs, a specific CSR)</td>
</tr>
</tbody>
</table>

A discovery request may be sent to query a CSx to discover the current state of a previously discovered CSR.

A.1.2 CSR Discovery Response

Editors Note: This section needs to be re-written based off of the Computational Storage API specification

An example CSR Discovery Response is shown in XXX. The response depends on the CSR filter in the corresponding CSR Discovery Request.

A CSR discovery response contains one entry for each resource that is discovered. Each CSR data structure may contain the following information:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSx Identifier</td>
<td>Unique identifier of the CSx target</td>
</tr>
<tr>
<td>CSR Identifier</td>
<td>Unique identifier of the CSR</td>
</tr>
<tr>
<td>CSR Vendor</td>
<td>Unique identifier of the CSR Vendor</td>
</tr>
<tr>
<td>CSR Type</td>
<td>Indicates the type of the CSR (e.g., CSE, CSF, CPU, Memory)</td>
</tr>
<tr>
<td>CSR Subtype</td>
<td>Indicates a subtype of the CSR</td>
</tr>
<tr>
<td>CSR State</td>
<td>Indicates the state of the CSR: (e.g., initializing, ready, configuring, available, busy, error)</td>
</tr>
</tbody>
</table>
### CSR Reservation
- Indicates if the CSR is reserved by a host: (Empty or Host ID)

### CSR Active Configuration Descriptor
- A CSR-specific data structure that indicates the currently active configuration, including options and parameters

### CSF Configuration Schema
- A self-describing data structure that indicates valid configurations, options and parameters for the CSF

### CSF Error
- Indicates if there are any CSF errors

### A.1.3 CSF Configuration Request

**Editors Note:** This section needs to be re-written based off of the Computational Storage API specification. This becomes the following sections: Resource allocation; CSF programming or downloading function; and CSF configuration.

A CSF configuration request contains the following information:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host Identifier</td>
<td>Globally uniquely identifier for the host</td>
</tr>
<tr>
<td>Host Credentials</td>
<td>Allows the host to authenticate with the CSF</td>
</tr>
<tr>
<td>CSx Identifier</td>
<td>Unique identifier of the CSx target</td>
</tr>
<tr>
<td>CSF Identifier</td>
<td>Indicates which CSF is being configured</td>
</tr>
<tr>
<td>CSF Configuration</td>
<td>Configuration information, as defined by the CSF configuration descriptor</td>
</tr>
</tbody>
</table>

The CSF Configuration includes configuration information, pointers to configuration information, and/or operational parameters.

### A.1.4 CSF Configuration Response

**Editors Note:** This section needs to be re-written based off of the Computational Storage API specification

A CSF configuration response contains the following information:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>Mandatory</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSx Identifier</td>
<td>String</td>
<td>Unique identifier of the CSx target</td>
<td>Yes</td>
</tr>
</tbody>
</table>
CSF configuration may take some time. The Host must either poll by sending discovery requests or have asynchronous notification to find out when the configuration is complete.

Newly created CSFs as a result of programming a CSE with a CSEE than has a attached CSFs are discovered using the standard CSF discovery process as described in 4.2.
Annex B. (Informative) Illustrative Examples

B.1 CSFs on a Large Multi-Device Dataset using Ceph

B.1.1 Introduction

A large multi-device dataset is a dataset that:

1. may not fit into a single storage device;
2. may be large enough to require hundreds or thousands of devices; and
3. may require scalable performance by utilizing many storage devices.

Due to the size of these datasets, a single server may be insufficient to house all of the necessary storage devices. Consequently, these datasets may also span servers. This illustrative example uses TCP/IP as it provides scaling for a large number of devices.

A large dataset is to be sharded into chunks, that have semantic meaning to the application and are stored across a set of storage devices. To act on that data in the computational storage sense it is necessary to map the data shards to the devices where they are stored and then deliver a function to each of the devices. That function is then able to be executed in the device against each of the data set’s shards stored in that device. This may be done simultaneously on thousands of devices.

There are many systems that enable the scaling of storage to thousands of devices. One such system used for this example is Ceph. Ceph allows many applications to jointly share shards of data called objects across potentially thousands of devices. Ceph is responsible for mapping the location of each of the objects across all the devices. Although intermediary servers called object storage daemons (OSDs) use local storage interconnects, the primary application interconnect is TCP/IP. Applications locate and interact with an object by a unique key that translates to a unique IP and TCP port address. Applications do not dictate this address but rather let Ceph manage the location of the object, abstracting the clients from the actual location.

Below is a diagram of Ceph showing client applications running in containers (App CT) using a variety of APIs (File, Block, S3) that are all implemented using the underlying Ceph RADOS API. This API permits the storing and retrieving of arbitrary sized objects as well as executing methods against objects.
The Ceph OSD servers in the diagram are responsive to the application's object requests. Although a single OSD satisfies a single object request, a dataset may be sharded into many objects and those objects will be stored across all of the available OSDs. Since the application is abstracted away from an object’s location, the application should also be abstracted away from the location of the execution environment. Furthermore, operating on a large dataset means engaging with many OSDs so abstracting the workload above the device level is also appropriate (i.e., an application should be able to have a distributed dataset while still operating on it as a single dataset).

Looking at the Computational Storage Architecture, Ceph could be viewed as the CSA and the OSDs could be viewed as Computational Storage Processors.

A CSE would be discovered and configured in the OSD.
However, nothing in the software architecture prevents running a Ceph OSD as a single device server. In this case, the OSD server would be viewed as a Computational Storage Drive (CSD).

B.1.2 Theory of Operation

In this section, we walk through the Large Multi-Device Dataset Ceph Computational Storage Theory of Operation (See Section 4) and apply it in a specific manner to this illustrative example. The three main phases of operation are:

a) Discovery (see section 4.2)
b) Configuration (see section 4.3)
c) Usage (see section 4.4)

B.1.3 Discovery

Most of the device discovery in this example is handled by Ceph; however, discovery of the Ceph system and the available CSRs and CSFs needs to be done. Built into Ceph is the ability to run code in the OSDs against an object by means of an object class definition. This facility should be viewed as the CSE. An object class definition contains methods that can be executed within an OSD against an object. An object class should be viewed as a CSF. Both preloaded and downloaded CSFs are able to be created, so discovery of these object classes will be necessary.
B.1.4 Configuration

A Ceph system shall be preinstalled and configured. The workloads are deployed by delivering the CSFs (i.e., object classes, if required) to each of the CSEs (i.e., OSD servers).

B.1.5 Usage

The application calls the CSF using the CSF parameters. Abstracted from the application, the CSF is translated into the Ceph object class. The CSF parameters are the object class method, the name(s) of the targeted objects, and any inbound parameters. The invocation of the CSF is handled by the Ceph RADOS rados_exec() function\(^1\). Upon return any outbound parameters are returned including the calls status.

\(^1\) There are several variants of the rados_exec() call, including versions that are associated with a read and write. There are also async versions of each call.
B.1.6 Example Application Deployment

Skyhook is an opensource middleware application that provides a Ceph backend to PostgreSQL. It enables PostgreSQL table stores by sharding tables on row boundaries, placing N rows of a table into a single object. If \( N = 1000 \) and an example table had a total of 100,000 rows, Skyhook would create 100 objects that would be distributed to potentially 100 OSDs/Devices in Ceph.

![Skyhook Example 1 Diagram]

Skyhook has defined its own object class (CSF) on each Ceph OSD node (CSE). This object class implements methods that evaluate SQL queries. When PostgreSQL receives a user submitted SQL query, it submits the query to Skyhook. Skyhook then submits that query to each of the 100 objects of the table in parallel by calling the RADOS exec() function (CSF Command). This call (CSF command) passes the query to the SQL evaluation method, each object independently evaluates the query for its rows, returning the results back to skyhook. Skyhook then assembles all of the results into a single result that is then returned to PostgreSQL.
Not only does this allow the DB to scale the performance and size independent of device capacities, but it also implements parallel execution.
B.2 Using a Containerized Application within Linux (CSEE with included CSF)

This illustrative example is of a Computational Storage Drive (CSD) based on the NVMe™ over PCIe® specification, that consists of one typical LBA based NVM storage device, multiple programmable applications processors co-located on the same controller capable of running Linux OS capable of running containerized application for searching data.

Specific assumptions for this illustrative example include:

  a) A server running a modern Linux kernel and user-space distribution, with an Intel Xeon processor;
  b) A single-ported single host system (i.e., no consideration for multi-port NVMe devices;
  c) No use of virtualization;
  d) A pre-activated CSEE (Running a Linux OS - CSEE1) is within the CSE;
  e) A CSEE is available to store in the Device Storage that is a Container (CSEE2) with a CSF1;
  f) CSF1 runs an Artificial Intelligence (AI) Application;
  g) No Peer-to-Peer capabilities or namespaces are used; and
  h) Leverages existing PCIe and NVMe methods around security.

B.2.1 Theory of Operation

In this section, we will walk through the Theory of Operation (see section 4) and apply it in a specific manner to this illustrative example. The three main phases of operation are:

  a) Discovery;
  b) Configuration; and
  c) Usage.

In this illustrative example, it is assumed that some updates to the existing NVMe Linux driver are made. The driver is required to be capable of recognizing and configuring a Computational Storage drive.

B.2.2 Discovery

NVMe/PCIe already has a very robust controller discovery and creation process. A PCIe device (i.e. an NVMe™ controller) is able to be discovered by the host at power-up time via PCIe bus enumeration or at run-time via a hot-plug event and bus rescan. A modern Linux operating system is able to detect PCIe devices via both the methods mentioned above, allowing discovery of a new NVMe controller to be done at any time in a running system.

Once an NVMe controller has been detected, the NVMe Computational Storage driver is able to be used to discover the capabilities of the controller via the NVMe Identify Admin command. The procedure is:

  1. PCIe enumeration discovers the NVMe controller of the CSD;
2. The Computational Storage NVMe Linux driver binds to this PCIe device; and
3. The driver discovers the capabilities of the Computational Storage Drive:
   a. An activated CSEE within the CSE that runs a Linux OS Environment (CSEE1).

![Diagram](image)

**Figure B.2.6 – CSx discovery process**

**B.2.3 Configuration**

The CSEE1 is active within the CSE. As shown in Figure B.2.7, the following steps take place:

A) the CSEE2 for the Docker Container is moved into the Device Storage of the CSD.  An Admin command causes the application processor(s) to boot into a Linux Environment. The software that supports Docker is loaded automatically as part of Linux start-up process; and

B) once Linux has booted, the CSEE2 for the Docker Environment is loaded with the CSF1.
The Computational Storage NVMe driver is notified so that the specific compute workload can be downloaded.

Figure B.2.7 – CSx Configuration process
B.2.4 Usage

Once the configuration process has completed and both CSEE1 and CSEE2 are activated and the CSF1 is ready to process data, the usage process steps (as shown in Figure B.2.8) can occur:

1) The CSF1 is told by the host to execute the function on data;
2) Input Data is pulled from Device Storage into Device Memory, in the Allocated Function Data Memory (AFDM) space of the Function Data Memory (FDM);
3) The data is operated on by the CSF1 and is then stored as Output Data in the Device Storage at the location specified by the function; and
4) The host can then access both the original Input Data and the new Output Data as required to complete the process steps.

Figure B.2.8 – CSx usage
B.2.5 Example of Application Deployment

The open source application called Openalpr (Open source Automatic License Plate Recognition) may be deployed using the Docker marketplace.

OpenALPR is an open source *Automatic License Plate Recognition* library written in C++ with bindings in C#, Java, Node.js, Go, and Python. The library analyzes images and video streams to identify license plates. The output is the text representation of any license plate characters. Openalpr provides a set of shared libraries but also makes use of a few other shared open source libraries.

In the Docker marketplace, there is a Docker image based on the “Vendor OS Choice” that contains the Openalpr shared libraries, its command-line utility application, and all the required shared libraries (e.g., OpenCV, python, and java)

In this specific example, the user would:

1) ssh from the host to the device using the default IP/username/password provided by the *device* vendor. (e.g., `user@server:~# ssh csd@ipaddress`)

2) Build a Docker image (i.e., `user@server:~# docker build -t openalpr https://github.com/openalpr/openalpr.git`)

3) Download test image (i.e., `user@server:~# wget http://plates.openalpr.com/h786poj.jpg`)

4) Run alpr on image (i.e., `user@server:~# docker run -it --rm -v $(pwd):/data:ro openalpr -c eu h786poj.jpg`)
The output of this example is:

```
user@server:~# /openalpr$ alpr ./h786poj.jpg

plate0: top 10 results -- Processing Time = 58.1879ms.
- PE3R2X    confidence: 88.9371
- PE32X     confidence: 78.1385
- PE3R2     confidence: 77.5444
- PE3R2Y    confidence: 76.1448
```
B.3 Data Deduplication CSF

B.3.1 Overview

Data deduplication is a technique used to reduce the amount of data stored on a storage device. Compression results in the removal of repeating bytes or streams within a chunk or segment of storage, but data deduplication results in the removal of matching chunks or segments of storage.

**Simplified Data Deduplication Process**

<table>
<thead>
<tr>
<th>Data Object / Stream</th>
<th>Data subject for deduplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C D A E B E D</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Chunking:</strong> Data is split into chunks.</td>
</tr>
<tr>
<td></td>
<td><strong>Processing:</strong> Chunks are identified and compared.</td>
</tr>
<tr>
<td></td>
<td><strong>Consolidation:</strong> Duplicate chunks are re-referenced so multiple pointers reference a single unique chunk. Redundant chunks are then released.</td>
</tr>
</tbody>
</table>

![Figure B.3.1 – Simplified Data Deduplication Process](image)

Figure B.3.1 describes a simplified data deduplication process where a given data object or stream is given to the deduplication process. The data object is split into chunks where the chunks can then be identified and compared. A location repository of pointers referencing the unique chunks is created and any duplicate chunks are released from the data object so that the resulting storage is reduced.

There are two generic ways of performing data deduplication on a device. The first way is post process data deduplication where the data that is being deduplicated already resides on the device and is processed at a scheduled time and duration. The second way is inline data deduplication where the data that is being deduplicated is immediately processed for deduplication so that only unique data is stored on the device.
Post Process Data Deduplication

1. Data object or stream is saved directly to the device.
2. The data is chunked with metadata and a repository is created to identify each chunk.
3. Chunks are compared. The chunk repository and chunk metadata are updated if chunks are identical.
4. The identical chunks are freed so that there is only a single instance of a chunk that was originally stored on the device.
5. The chunks are then consolidated within the device.

Figure B.3. 2 Post Process Data Deduplication

Figure B.3. 2 describes post process data deduplication. This process does not interfere with the initial ingestion of the data object or stream and allows the system to schedule a time and duration for the deduplication process. Once the data object or stream is stored, the data is chunked where the chunk metadata describes the size, references, and location of the data. A repository is also created to map the location of each chunk with respect to the way the data was initially stored. The chunks are compared and if chunks are identical, the chunk metadata of the first instance of the chunk is updated with a reference count. The repository pointer for the identical chunk is also updated to the first instance of the chunk. Once the data object or stream has had all the chunks compared, the identical chunks are freed so that only a single instance of the chunk is stored on the device. Finally, the chunks are consolidated within the device.
Figure B.3. 3 Inline Data Deduplication

**Figure B.3.3** describes inline data deduplication. This process interferes with the ingestion of the initial data object or stream being saved to the device. As the data object or stream is being written to the device, chunks are compared and written only once if identical. This allows for less storage space to be used on the initial ingestion of the data object or stream. The first time the data is stored into the device, a chunk repository is created to describe the location of the chunks within the data object or stream. The initial chunk is stored with chunk metadata that describes size, location, and references to the chunk. Subsequent chunks are then compared with existing chunks, and if unique, it is stored as a unique chunk. If it’s identical, the repository is updated to reference the single instance of the chunk and the number of references in the chunk metadata is updated. The process continues until the data object or stream is complete.

These descriptions of data deduplication are all simplified to explain generic ways to perform data deduplication. The CSx that will implement data deduplication will likely have more intricate proprietary ways of performing the data deduplication.

**B.3.2 Theory of Operation**

To successfully implement data deduplication as a CSF, a CSx must be able to communicate the ability to perform data deduplication within a CSE. Once the ability is determined, the CSF then needs to be successfully configured. The data object or data stream can then be processed by the CSE using the data deduplication CSF while allowing monitoring of the progress of that operation.
This illustrative example will attempt to provide the initial framework of the following steps needed to successfully allow for a CSx to perform data deduplication as a CSF.

B.3.3 Discovery

To determine if a CSx supports the Data deduplication CSF, the CSx needs to first be discovered as a CSx. If a CSx supports the Data deduplication CSF, then the CSx needs to also indicate if the Data deduplication CSF allows configuration. If the CSx allows for certain configuration parameters to the Data deduplication CSF, then the configurable parameters need to be shared. The following is a possible list of configurable parameters:

1. Supported Chunk Sizes – The acceptable size values to chunk the data object into comparable chunks. Typically, these are large, but they could also be variable.
2. Scheduling – The schedule and duration of post process data deduplication.
3. Failover – The action to take if data deduplication is interrupted (e.g., discarding, resuming from last good write, restarting).
4. Monitoring – The type of data to collect during data deduplication (e.g., the current data deduplication space savings, the I/O rate of the data deduplication operation, the size of the data processed, the size of the data remaining to be processed, the percentage complete).
5. Inline or Post Process Deduplication
6. Type of method to perform chunk comparison – Three common types are hashing, binary comparison and delta differencing.
7. Hashing Algorithm – Type of hashing algorithms allowed to identify unique chunks.
8. Data deduplication Analysis – Methods to determine the likely savings gained by performing data deduplication on a data object or stream.
9. Operational Interruption – Whether or not the data deduplication operation is able to be interrupted for purpose of either suspending, abandoning or resuming the request.

Note that discovery may also return the default values of the configurable parameters as well as capabilities like operational interruption that the Data deduplication CSF supports.

B.3.4 Configuration

Once a CSx that supports the Data deduplication CSF is discovered, the Data deduplication CSF can be configured if allowed. It’s possible that the Data deduplication CSF will have default values that the user may not want to override, and configuration is not necessary. If the user wants to configure the Data deduplication CSF, then the parameters returned by discovery need to be set and sent to the Data deduplication CSF before the data deduplication operation is performed by the CSE.

Table 1

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
</table>

---

**Computational Storage Architecture and Programming Model**

**Working Draft Version 0.8 rev 0**

53
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeduplicationSupport</td>
<td>Mandatory</td>
<td>Specifies whether Deduplication can be configured as a CSF within the CSE.</td>
</tr>
<tr>
<td>DeduplicationType</td>
<td>Optional</td>
<td>The type of deduplication to perform. This can be either inline or post process deduplication</td>
</tr>
<tr>
<td>DeduplicationSchedule</td>
<td>Conditional</td>
<td>Valid when post process deduplication is specified. This would be the time, frequency, and duration when the deduplication would be performed.</td>
</tr>
<tr>
<td>DeduplicationChunk</td>
<td>Optional</td>
<td>If set, the size of the chunks to perform deduplication comparisons.</td>
</tr>
<tr>
<td>DeduplicationFailover</td>
<td>Optional</td>
<td>If set, the action to take when data deduplication is interrupted. Possible actions are discarding, resume from last good write, restart, etc.</td>
</tr>
<tr>
<td>DeduplicationMonitoring</td>
<td>Optional</td>
<td>If set, type of data to collect during data deduplication. Possible types of data to collect are current data deduplication space savings, the I/O rate of the data deduplication operation, the size of the data processed, the size of the data remaining to be processed, the percentage complete, etc.</td>
</tr>
<tr>
<td>DeduplicationComparison</td>
<td>Optional</td>
<td>If set, type of method to perform chunk comparison. Three common types are hashing, binary comparison and delta differencing</td>
</tr>
<tr>
<td>DeduplicationHash</td>
<td>Optional</td>
<td>If set, type of hashing algorithm to identify unique chunks</td>
</tr>
<tr>
<td>DeduplicationSavings</td>
<td>Optional</td>
<td>If set, perform an analysis on data being received to determine the amount space saved.</td>
</tr>
<tr>
<td>DeduplicationOperation</td>
<td>Optional</td>
<td>If set, allow for the interruption of the data deduplication function by request with option to either suspend, abandoned or resume the data deduplication operation.</td>
</tr>
</tbody>
</table>

Figure B.3. 4 Configuration Parameters
Figure B.3.4 lists the type of parameters that could be configured for the Data deduplication CSF prior to sending a data object or stream to the CSx.

B.3.5 Operation

The data deduplication CSF will begin when the data object or stream is sent. The data object or stream may have already been processed by another CSF or be processed for input into another CSF. For example, data deduplication is inefficient on encrypted data, so the Encryption CSF should first decrypt the data before sending it to the data deduplication CSF. Additionally, once the data has been deduplicated, the data is able to then be sent to the compression CSF for additional storage savings.

The operation of the data deduplication CSF may be interrupted if allowed. Otherwise, the user will need to wait for the operation to complete or fail to determine the next course of action.

B.3.6 Monitoring

As the data object is being processed by the data deduplication CSF, the user can get status and statistics on the process. The following are possible status and statistics to monitor:

1. Current data deduplication space savings
2. The I/O rate of the data deduplication operation
3. Current amount of data processed by the data deduplication operation
4. Current amount of data remaining to be processed by the data deduplication operation
5. The percentage of completion of the data deduplication operation
6. Success or failure of the operation
7. Existing state of the operation such as paused, interrupted, resumed, etc.

Based on the status and statistics, the user can then determine if the operation needs to be paused, abandoned, or resumed.

B.3.7 CSF Data Deduplication Example
1. It is necessary to create a volume (see Figure B.3. 5 Volume Creation) to perform data deduplication.

![Figure B.3. 5 Volume Creation](image)

2. Configure data duplication to be enabled on the created volume by sending a configuration parameter to the data deduplication CSF. This operation is illustrated in Figure B.3. 6 Enable Deduplication.

![Figure B.3. 6 Enable Deduplication](image)

3. Verify that data deduplication is enabled and ready for the created volume by retrieving status. The status indicates that the configurable parameters have been set. This operation is illustrated by Figure B.3. 7 Retrieve Status.
4. Turn off the schedule of the post process data deduplication on the volume by sending the data deduplication CSF a request that data deduplication not be scheduled to occur on the volume. This is shown in Figure B.3. 8 Disable Deduplication Scheduling.

5. Mount the volume to a server and copy files from existing server directories into the new directory as shown in Figure B.3. 9 Copy files. This writes the files to the newly created
volume and since there is no deduplication scheduled, the data is not deduplicated.

6. Examine the volume for the storage consumed and space using existing server tools as shown in Figure B.3. 10 Verify space utilized. No space savings have been made since deduplication has yet to occur.

7. Send a request to the data deduplication CSF, to schedule the data deduplication immediately on the volume as shown in Figure B.3. 11 Schedule deduplication. The
data deduplication then occurs on the volume.

8. Monitor the progress of deduplication by sending the data deduplication CSF a status request. The data deduplication CSF returns the amount of space that has been deduplicated and the percentage complete. This is shown in Figure B.3. 12 Monitor Progress.

9. When the deduplication is complete, check the space savings by sending a status request to the data deduplication CSF as shown in Figure B.3. 13 verify space savings. The space savings depends on the amount of data that was deduplicated on the
volume.

![Diagram](image)

Figure B.3. Verify space savings

- Retrieve Data
- Deduplication Status: 100% Complete
- Space Savings: 80%
- Host

Computational Storage Device (CSD)
B.4 A Computational Storage Function on a NVM Express and OpenCL based Computational Storage Drive Illustrative Example

B.4.1 CSEE Example

A PCIe OpenCL-based Computational Storage Drive (CSD) that consists of an NVMe Controller and an OpenCL accelerator (CSE) which can execute Computational Storage Functions (CSFs). In this example, the NVMe controller and OpenCL accelerator appear as two separate PCI physical functions (PF) on the host. The OpenCL accelerator exposes part of the Function Data Memory (FDM) over the PCIe BAR of its PCI physical function (PF) and this exposed memory is mapped into the host’s address space. This enables the host software to allocate buffers from this PCIe BAR memory and use them to move data between the NVMe controller and the OpenCL accelerator directly while bypassing the host system memory entirely. This feature is called PCIe peer-to-peer (P2P) transfer. For the rest of this illustrative example the term P2P is associated with the memory exposed over PCIe BAR and the direct transfer mechanism between the NVMe controller and the OpenCL accelerator. Note that the P2P memory is part of the FDM which is part of the computational storage resources (CSR) and not associated with NVMe storage controller within this CSD.

Other specific assumptions for this illustrative example include:

- An application class processor running a modern Linux kernel and user-space distribution.
- A single host system. i.e. no consideration for multi-port NVMe devices.
- No virtualization.
- Leverage existing PCIe and NVMe methods around security.
Figure B.4.1 CS architectural diagram adapted for this illustrative example

B.4.2 Computational Storage Drive

Figure B.4.1 describes the elements comprising a CSD. There is a Computational Storage Engine (CSE) combined with persistent memory data storage in the form of an NVMe SSD within a single device (represented by Storage Controller and Device Storage blocks in the figure). The CSEE provides an environment for executing the CSF. If the input data for the CSF is on the device storage, it can be directly transferred (using P2P transfer mechanism) to an AFDM buffer which is allocated from the P2P memory. Similarly, if the output data from the CSF is to be stored on the device storage, it is able to be directly transferred from the P2P AFDM buffer using P2P transfer mechanisms. This example also applies to a case with two discrete PCIe devices connected to the same PCIe fabric (i.e., they are not part of same device as shown in the above...
In this example the OpenCL device is referred to as a Computational Storage Processor (CSP) with an exposed PCIe P2P BAR memory and the second device is an NVMe SSD.

This example does not explicitly address multiple CSDs in one system, but the architecture, including the software drivers for OpenCL and NVMe all support multiple devices in one system.

B.4.3 Theory of Operation

This section walks through the Theory of Operation and applies it in a specific fashion to this illustrative example. The three main phases are:

- Discovery
- Configuration
- Usage

B.4.4 Discovery

In this illustrative example, the storage controller of the CSD is discovered and configured using the procedures described in the NVMe base specification without any changes. The OpenCL accelerator (CSE) is discovered and configured using the OpenCL APIs.

PCIe devices (NVMe controller and OpenCL device) are discovered by the host at power-up via PCIe bus enumeration or at run-time via a hot-plug event and bus rescan. A modern OS like Linux binds the relevant drivers to the discovered PCIe devices. In this case the NVMe driver is bound to the NVMe controller of the CSD and the OpenCL runtime driver is bound to the OpenCL device.

B.4.4.1 NVMe Function Discovery

Once an NVMe controller has been detected, the NVMe driver on the host discovers the capabilities of said controller via the NVMe Identify Admin command. NVMe has the concept of namespaces, which refers to discrete resources behind an NVMe controller and one controller is able to support many namespaces as well as being able to (optionally) create and delete namespaces (via namespace management commands). Therefore:

1) PCIe enumeration discovers the NVMe controller.
2) The NVMe driver binds to this PCIe device and controller capabilities and namespaces are discovered using standard NVMe controller Identify Admin commands.
3) The driver configures the NVMe namespaces using NVMe NS identify commands and makes them available to the storage stack on the host.

In this example the driver will discover one namespace on the NVMe controller of the CSD

a) Namespace 1: Conventional LBA based storage namespace.
At this point the information about the namespaces on this NVMe controller of the CSD is able to be displayed with admin tools like nvme-cli. As an example the output of nvme-cli list for such a device might look like:

<table>
<thead>
<tr>
<th>Node</th>
<th>SN</th>
<th>Model</th>
<th>Namespace Type</th>
<th>Format or SubType</th>
<th>FW Rev</th>
</tr>
</thead>
<tbody>
<tr>
<td>/dev/nvme0n1</td>
<td>nvme1</td>
<td>Vendor A 1</td>
<td>Conventional LBA</td>
<td>512 B + 0 B</td>
<td>1.0</td>
</tr>
</tbody>
</table>

B.4.4.2 OpenCL CSP Function Discovery

As the OpenCL accelerator is on a separate PCI physical function, the first two stages of discovery for the OpenCL CSE mirror those of the NVMe controller:

1. PCIe enumeration is used to discover the OpenCL CSP.
2. The Linux OpenCL Runtime driver binds to this PCIe device

The OpenCL Runtime driver implements platform-specific features that allow applications to query OpenCL devices, device configuration information, and to create OpenCL contexts using one or more devices.

The OpenCL runtime driver also discovers the PCIe P2P BAR memory exposed by the OpenCL CSE and maps it to the host’s physical address space.

OpenCL defines set of APIs to discover OpenCL platforms and devices and to configure them. In OpenCL, a platform is defined as the set of OpenCL devices from a vendor that implement OpenCL functionality. The platform vendor also provides the OpenCL runtime drivers to manage the OpenCL devices. A typical usage of OpenCL device involves discovering the platform of the vendor and discovering the devices on that platform.

The OpenCL Runtime driver implements APIs that allow applications to query OpenCL devices and device configuration information.

The APIs relevant to discovering the OpenCL device are described below:

This API returns the number and list of OpenCL platforms found on the system:

```c
cl_int clGetPlatformIDs(cl_uint num_entries, cl_platform_id *platforms, cl_uint *num_platforms)
```

This API is used to get the platform specific info of the desired vendor platform:

```c
cl_int clGetPlatformInfo(cl_platform_id platform,
                        cl_platform_info param_name,
                        size_t param_value_size,
                        void *param_value,
                        size_t *param_value_size_ret)
```
This API is used to find the number of OpenCL devices and their device-ids of the specific vendor platform selected.

```c
cl_int clGetDeviceIDs(cl_platform_id platform,
                      cl_device_type device_type,
                      cl_uint num_entries,
                      cl_device_id *devices,
                      cl_uint *num_devices)
```

Finally, the following API can be used to iterate over all the discovered devices and find the desired device based on parameter CL_DEVICE_NAME. In this example the OpenCL accelerator inside the CSD will have a unique name that can be matched.

```c
cl_int clGetDeviceInfo(cl_device_id device,
                      cl_device_info param_name,
                      size_t param_value_size,
                      void *param_value,
                      size_t *param_value_size_ret)
```

**B.4.5 Configuration – Explicit Mode**

OpenCL provides a rich set of APIs to configure the CSE. These include mechanism to partition a device, create contexts, download OpenCL kernels (CSF), allocate memory (FDM), and more.

**B.4.6 Usage – Storage Direct**

The OpenCL device can be further optionally partitioned to sub-devices each having its own context and command queue. The following API is used to create sub-devices.

```c
cl_int clCreateSubDevices (cl_device_id in_device ,
                           const cl_device_partition_property *properties ,
                           cl_uint num_devices ,
                           cl_device_id *out_devices ,
                           cl_uint *num_devices_ret )
```

Once the device is selected, an OpenCL context must be created on that device using the following API

```c
cl_context clCreateContext(cl_context_properties *properties,
                           cl_uint num_devices ,
                           const cl_device_id *devices,
                           ...)
```
void *pfn_notify (  
    const char *errinfo,  
    const void *private_info,  
    size_t cb,  
    void *user_data  
),  
void *user_data,  
cl_int *errcode_ret)

The OpenCL context can be equated with the CSEE in the CSD. The context is used to create a command queue between the host and the device and to load OpenCL kernels, allocate memory objects.

The API to create the command queue is:

cl_command_queue clCreateCommandQueue(cl_context context,  
    cl_device_id device,  
    cl_command_queue_properties properties,  
    cl_int *errcode_ret)

After the command queue is created, an OpenCL program is able to be created from a binary file (whose format is specific to the OpenCL device execution environment). The OpenCL program may contain one or more OpenCL kernels. An OpenCL kernel is equivalent to a CSF on the CSD.

The following APIs are used to build the OpenCL program from binary and create OpenCL kernels (CSFs)

cl_program clCreateProgramWithBinary (   cl_context context,  
    cl_uint num_devices,  
    const cl_device_id *device_list,  
    const size_t *lengths,  
    const unsigned char **binaries,  
    cl_int *binary_status,  
    cl_int *errcode_ret)

cl_kernel clCreateKernel ( cl_program program,  
    const char *kernel_name,  
    cl_int *errcode_ret)

The input and output buffers for the OpenCL kernels are allocated on the device and mapped to host side buffers using the following APIs

The following API allocates a buffer on the OpenCL device and a corresponding buffer on the host memory

cl_mem clCreateBuffer (cl_context context,
The following API returns the host mapped address of the buffer created in the previous API.

```c
void * clEnqueueMapBuffer (cl_command_queue command_queue,
                            cl_mem buffer,
                            cl_bool blocking_map,
                            cl_map_flags map_flags,
                            size_t offset,
                            size_t cb,
                            cl_uint num_events_in_wait_list,
                            const cl_event *event_wait_list,
                            cl_event *event,
                            cl_int *errcode_ret)
```

The device side buffer handle returned by clCreateBuffer APIs is passed to OpenCL kernel using the following API

```c
cl_int clSetKernelArg (cl_kernel kernel,
                        cl_uint arg_index,
                        size_t arg_size,
                        const void *arg_value)
```

There is no explicit configuration required for the NVMe controller as the host driver already enumerates and configures the namespaces during the discovery process as per the NVMe standard

### B.4.7 Usage – Explicit Mode Computational Storage

In this example we look at a simple CSF use case where data stored on the storage of the CSD is read, a transformation is performed on that data and the result is returned to the host.

As the input data is stored on the NVMe data storage of the CSD, it needs to be first read and given to the CSF. There are two ways to do this: The host issues an NVMe read operation and reads the data into its memory. Once that is complete, moves that data to OpenCL device memory (AFDM) using the OpenCL API clEnqueueMigrateMemObjects. The other method is to make use of the P2P mechanism. In this mechanism the host issues NVMe read operation but the destination of that operation is the P2P memory on the OpenCL device. Using the PCI P2P operation, the NVMe controller directly moves the data into OpenCL device memory (AFDM). Using the P2P mechanism reduces the extra data movement from NVMe storage to host memory and then back to OpenCL device memory (AFDM).
Once the data is available in AFDM, the OpenCL kernel (CSF) can be triggered to run using the following API.

```c
cl_int clEnqueueTask (cl_command_queue command_queue, 
    cl_kernel kernel, 
    cl_uint num_events_in_wait_list, 
    const cl_event *event_wait_list, 
    cl_event *event)
```

The host now waits for the CSF to complete its operations using the API

```c
cl_int clWaitForEvents( 
    cl_uint num_events, 
    const cl_event* event_list)
```

Once the operation is complete, the result data can be moved back to host memory using the OpenCL API `clEnqueueMigrateMemObjects` API

```c
cl_int clEnqueueMigrateMemObjects (   cl_command_queue command_queue, 
    cl_uint num_mem_objects, 
    const cl_mem *mem_objects, 
    cl_mem_migration_flags flags, 
    cl_uint num_events_in_wait_list, 
    const cl_event *event_wait_list, 
    cl_event *event )
```

If the use case is to store the result of the CSF back to the NVMe storage of the CSD, the output buffer also can be allocated from the P2P area in the AFDM and NVMe write operation can be initiated to move it into the storage directly using the P2P transfer mechanism between NVMe controller and the OpenCL device.
B.5 Data Compression CSF Example

B.5.1 Overview

Data compression aims to reduce the amount of data stored on a storage device. By reducing the amount of data being physically written to or read from storage media (e.g., flash memory chips), data compression can also improve the storage device IO performance and lifetime span, in addition to storage cost saving.

Different from off-loading compression/decompression through a specific API, Data Compression CSF carries out compression/decompression on the IO data path, transparently from the host (i.e., host simply issues normal IO write/read requests to a CSD with a Data Compression CSF, without calling any other specific API). In order to materialize the storage cost reduction, Data Compression CSF must expose a logical storage space that is larger than its internal physical storage space. However, due to the runtime variation of data compressibility, it is possible that the physical storage space will be used up before the logical storage space is used up. Moreover, it is desirable for users to know how well different files or objects are compressed by Data Compression CSF. Therefore, Data Compression CSF must provide adequate reporting mechanism and observability in terms of data compression.

B.5.2 Theory of Operation

In order to successfully implement data compression CSF that operates on all write data, a CSx must be able to communicate that it contains a CSE that is able to perform data compression CSF. Once the CSF’s ability is determined, the CSF should allow the host to monitor and query the effect of data compression. This illustrative example will attempt to provide the initial framework of the following steps needed to successfully allow for a CSx to perform a data compression CSF.

B.5.3 Discovery

In order to determine if a CSx supports the Data Compression CSF, the CSx must first be discovered as a CSx, with a CSE that is able to perform the function. If the CSE is able to perform a Data Compression CSF, then the CSE must also indicate if the Data Compression CSF allows configuration. If the CSE allows for configuration parameters to the Data Compression CSF, then the configurable parameters must be shared. The following is a possible list of configurable parameters:

1. Supported compression block sizes – The acceptable values of compression data block size. The block size can be set globally for the entire Data Compression CSF, be set separately for each logical storage space region of the Data Compression CSF, or even be set for each individual write request sent to the Data Compression CSF.
2. Supported maximum logical storage space – The maximum size of the logical storage space that can be exposed by the Data Compression CSF.
3. Monitoring – The type of information to collect during runtime operation. Such information can include the runtime physical storage capacity usage of the entire Data Compression CSF, and the runtime physical storage capacity usage of any given logical storage space.

Note that discovery may also return the default values of the configurable parameters.

B.5.4 Configuration

Once the configurable parameters of a CSE that supports the Data Compression CSF are discovered, the Data Compression CSF is able to be configured if allowed. It is possible that the Data Compression CSF will have default values that the user may not want to override, and configuration is not necessary. If the user wants to configure the Data Compression CSF, then the parameters returned by discovery need to be set and sent to the Data Compression CSF before the data compression operation is performed.

B.5.5 Monitoring

As the data is being written to the Data Compression CSF, the user can get statistics on the compression. The following are possible statistics to monitor:

1. Current physical storage space usage of the entire CSF
2. Current physical storage space usage of any logical storage space region of CSF
3. Lifetime data compression ratio of all the data written to CSF so far
4. Data compression ratio over a specified amount of data that has been written to the CSF
B.6 Data Filter CSF Example

B.6.1 Overview

As one important category of operations in data analysis, data filtering aims to filter out the data that are not needed by a query. In conventional practice, a CPU (or GPU) is responsible for data filtering, which requires transferring all the raw data from the storage device into the CPU (or GPU) memory. The unique feature of Data Filter CSF is to pushdown the data filtering operation from the CPU (or GPU) to a CSE in the CSD. It can offload the data filtering operation from the CPU (or GPU), leading to higher system performance, and less host resource contention in terms of CPU/GPU cycles, memory capacity and bandwidth, and I/O bus bandwidth. The following example illustrates using data filter CSF to carry out in-storage data filtering, as shown in Figure 1. A table with four columns is stored in the Data Filter CSF that receives a request “SELECT ID where State=CA” that seeks the IDs of all the table entries in which State equals to CA. As illustrated in the figure, the Data Filter CSF fetch all the table entries from the storage media, extracts each table entry, and checks whether the 4th column in the entry equals to CA. After scanning the entire table, the Data Filter CSF returns the IDs of all the matching entries.

Figure B.6. 1 An example to illustrate the function of data filter FCSS

B.6.2 Theory of Operation
In order to successfully implement data filter as a CSF, a CSx must be able to communicate that it contains a CSE with the ability to perform data filter CSF. Once the ability is determined, the CSF should allow the host to query the supported filter functions and data schema. This illustrative example provides the initial framework of the following steps needed to successfully allow for a CSE to perform data filter as an CSF.

B.6.3 Discovery

The steps in discovery are:

a) Discover if the Data Filter CSF allows configuration.
   a. If the Data Filter CSF allows configuration, then discover the configurable parameters. Discovery may also return the default values of the configurable parameters.
   b. The following is list of possible parameters:
      i. Supported data formats (e.g. Parquet/ORC, JSON, XML formats). In order to perform data filtering, Data Filter CSF must be able to understand the data format specified by the host.
      ii. Supported data types (e.g., ASCII string, integer)
      iii. Supported filtering operations (e.g. >, <, =)
      iv. Failover – The action to take (e.g. rollback to host-based data filter) if data filter is interrupted.

b) Discover operational attributes of the Data Filter CSF
   a. Monitoring Capabilities – The type of process information that can be collected during data filtering. Such possible information includes the I/O rate of the data filtering operation, the size of the data processed, the size of the data remaining to be processed, the percentage complete, etc.
   b. Operational Interruption Capability – Whether or not the data filtering operation can be interrupted for purpose of either suspending, abandoning, or resuming the request.

B.6.4 Configuration

Once a CSE that supports the Data Filter CSF is discovered, the Data Filter CSF is able to be configured if allowed. It is possible that the Data Filter CSF will have default values that the user may not want to override, and configuration is not necessary. If the user wants to configure the Data Filter CSF, then the parameters returned by discovery need to be set and sent to the Data Filter CSF before the data filtering operation is performed.

B.6.5 Operation

To utilize data filter CSF to carry out in-storage data filtering, host passes enough information about the data filter operation to the CSE that executes the data filter CSF. Accordingly, the data filter CSF performs in-storage data filtering and returns the results back to the host. The information about the data filter operation may include:
1. The address of to-be-processed data
2. The data format (e.g., MySQL, Parquet) and schema (e.g., the number of columns in the table, and data type of each column)
3. The specific filter operation to be performed
4. The host memory address for the returned data

B.6.6 Monitoring

As the data object is being processed by the data filter CSF, the user can get status and statistics on the process. The following are possible status and statistics to monitor (not all items are required; not all items are unique to this CSF):

8. The I/O rate of the data filter operation
9. Current amount of data processed by the data filtering operation
10. Current amount of data remaining to be processed by the data filtering operation
11. The percentage of completion of the data filtering operation
12. Success or failure of the operation
13. Existing state of the operation such as paused, interrupted, resumed, etc.

Based on the status and statistics, the user can then determine if the operation needs to be paused, abandoned or resumed.
B.7 Scatter Gather CSF

B.7.1 Overview

Many computational storage use cases require offloading and coordinating data movement, including data flows between multiple CSFs. Offloading data movement has long-standing precedents in computer architectures, specifically, Direct Memory Access (DMA).

Use cases for DMA in computational storage include:

a. Host to CSx: Gathering a series of blocks of data from host memory, which are potentially scattered across host memory space, to be transferred to a CSx (this use case is already implemented for NVMe-based CSx via PRP/SGLs);
b. CSx to Host: Scattering a series of blocks of data from a CSx to be transferred into host host memory, where destination addresses are potentially scattered across host memory space (this use case is already implemented for NVMe-based CSx);
c. Peer-to-Peer: Scatter-Gather between CSxes without having to traverse host memory (this use case is partially implemented for some NVMe-based CSx);
d. Fanout: Scatter-Gather to multiple CSxes; and
e. Pipelining: Managed flows between three or more devices, including flow control.

All of these functions need to be provided such that they can be performed to and across multiple supported fabrics.

In addition to these data movement functions, there is also a need to provide a standardized:

a. representation for applications to specify desired data movements;
b. representation for application to specify and compose data pipelines;
c. mechanism to handle private and internal busses and devices;
d. mechanism to handle flow control and back-pressure;
e. mechanism to handle buffer management for transformational processes where the size of output data changes from the size of input data; and
f. mechanism to handle errors and failure recovery.

In addition to host-managed DMA, there is a need for a FCSS to provide and coordinate DMA between CSxes.

B.7.1.1 Representation of a Data Movement Request (DMR)

For an application to express desired data movement in an interoperable way, a standardized representation of the desired data movement is required. This illustrative example proposes use of CBOR as a compact binary representation of a JSON-based data structure that describes a desired data movement request (DMR).

For readability, this document will use the corresponding JSON representation of the CBOR data structure.
Once constructed by an application, a DMR can be translated into a fabric-specific data movement operations, decomposed into topology-specific flows, and offloaded to DMA-capable CSxes.

```json
{
    "ep": {
        "1": {
            "etype": "NVMe"  
        }
    },
    "flows": [
        {
            "type": "memory"  
        }
    ]
}
```

Figure B.7. 1 Top-level DMR structure

**B.7.1.2 Endpoints**

Endpoint definitions specify the entities involved in the data movement request. These can include host memory, fabric-accessible CSx memory, storage devices, internal (non-fabric accessible CSx memory), etc.

```json
"ep": {
    "1": {
        "etype": "NVMe"  
    },

    "ncuid": "144D..."  
}
```

Figure B.7. 2 Example endpoint definition structure for an NVMe endpoint

This example endpoint uniquely specifies an endpoint named “1”, which is an NVMe namespace on a given NVMe controller, including the type of the namespace. Other endpoints include host memory (“mem”), which allows for a base address to be specified.

Endpoints also specify the addressing mode. The following address types are anticipated:

**Memory:** A contiguous range of memory addresses space

**Block:** A contiguous range of fixed-length blocks of data within an address space

**Key:** The contents of a given key in a key/value space

Endpoints within internal devices and busses can also be specified when using a scatter-gather CSS to offload data movement.
B.7.1.3 Flows

Flow definitions specifies what data should be moved between endpoints.

```
{
    "src": [
        <One or more source locations>
    ],
    "dst": [
        <One or more destination locations>
    ]
}
```

Figure B.7. 3 Flow definition structure

Source and destination locations are expressed as <endpoint, start-address, length> triples. The address is dependent on the endpoint type, and when an endpoint defines a default block size, the length can be omitted if the default block length is to be used.
B.7.1.4 Example DMRs

The following example DMRs illustrates some of the use cases:

```json
{
  "ep": {
    "1": {"eptyp": "mem",
           "mbadr": "00002020FFF00000"},
    "2": {"eptyp": "NVMe",
           "ncuid": "2366......",
           "nnuid": "ACDE48234567ABCD",
           "nntyp": "lba"},
    "3": {"eptyp": "NVMe",
           "ncuid": "53F2......",
           "nnuid": "00A0BF398F8912AA",
           "nntyp": "lba"},
    "4": {"eptyp": "NVMe",
           "ncuid": "2354......",
           "nnuid": "00A0BF3923DS9823",
           "nntyp": "lba"}
  },
  "flows": [ {
    "src": [ ["1", "0", "2048"], ["1", "6144", "2048"], ["1", "2048", "2048"], ["1", "8192", "2048"], ["1", "4096", "2048"], ["1", "10240", "2048"]
    ],
    "dst": [ ["2", "4214"], ["3", "4214"], ["4", "4214"]
    ]
  }
}
```

Figure B.7. 4 Scatter DMR
This example scatters data from a BAR to three NVMe namespaces, as illustrated in figure B.7.5:

```json
{
    "ep": {
        "1": {
            "eptyp": "mem",
            "mbadr": "00002020FFF00000"
        },
        "2": {
            "eptyp": "css",
            "cssID": "273846333"
        },
        "3": {
            "eptyp": "css",
            "cssID": "981234784"
        },
        "4": {
            "eptyp": "NVMe",
            "ncuid": "2366......",
            "nnuid": "ACDE48234567ABCD",
            "nntyp": "lba"
        },
        "5": {
            "eptyp": "NVMe",
            "ncuid": "53F2......",
            "nnuid": "00A0BF398F912AA",
            "nntyp": "lba"
        },
        "6": {
            "eptyp": "NVMe",
            "ncuid": "2354......",
            "nnuid": "00A0BF3923DS9823",
            "nntyp": "lba"
        }
    },
    "flows": [
        {
            "src": [
                "1", "0", "4096000"
            ],
            "dst": ["2"
            ]
        },
        {
            "src": [
                "2"
            ],
            "dst": ["3"
            ]
        },
        {
            "src": [
                "3"
            ],
            "dst": [
                ["4", "4214", "20480000"],
                ["5", "4214", "20480000"],
                ["6", "4214", "20480000"
            ]
        }
    ]
}
```

Figure B.7.5 Example Scatter DMR

Figure B.7.6 Pipeline DMR
This example shows a pipeline where data is streamed from host memory to a first CSS (compression), streamed to a second CSS (erasure coding), then scattered across three NVMe namespaces, as illustrated in the below diagram:

![Diagram](image.png)

**Figure B.7. 7 Example Pipeline DMR**

### B.7.2 Scatter-Gather CSS Theory of Operation

While the above examples can all be coordinated by the host, it is often advantageous to offload data movement to a CSx. This is implemented by a Scatter-Gather CSS (SG CSS).

This illustrative example will attempt to provide the initial framework of a FCSS that implements scatter-gather.

#### B.7.2.1 Discovery

In order to determine if a CSx provides a SG FCSS, the CSS must first be discovered. The discovery process will also return configuration and status information as specified in the normative sections of this specification.

At this time no configurable parameters are anticipated.

Note that discovery may also return the default values of the configurable parameters as well as a list of private endpoints that the Scatter-Gather FCSS can address.

#### B.7.2.2 Configuration

Once a Scatter-Gather FCSS is discovered, it can be configured. At this time no configurable parameters are anticipated.
B.7.2.3 Operation

The Scatter-Gather FCSS will begin processing a DMR when the DMR CBOR is sent to the CSS’s NVMe input queue, or written into PCIe memory (if it is not NVMe-based). When the DMR is received, it is validated and executed, moving data as requested.

The operation of the Scatter-Gather FCSS may be interrupted if allowed. Otherwise, the user will need to wait for the operation to complete or fail to determine the next course of action.

B.7.2.4 Monitoring

As the DMR is being processed, the user can get status on the data movement. The following are possible status to monitor:

a. Flow progress; and
b. Error status.

Based on the status, the user can then determine if the operation needs to be paused, abandoned or resumed.