Sunfish OpenFabrics Management Framework for Composable Disaggregated Infrastructures

Version 0.3

ABSTRACT: Sunfish is designed for managing composable disaggregated resources over multiple fabrics using a central repository and an open-source API and toolset. Sunfish is designed for manipulating connected hardware resources using client-friendly RESTful abstractions and configuring fabric interconnects so that datacenter and AI workloads can be linked with available resources over dynamic fabric infrastructures.

The Sunfish OpenFabrics Management Framework API defines a RESTful interface and a standardized data model to provide data structures to help simplify the development of composable distributed, disaggregated, computer architectures. Sunfish contains abstract data structures that represent computer system resources, available network fabric components and management, current resource operational conditions, and abstracted representations of composed disaggregated computing systems.

Last Updated 04/23/2024

USAGE

Permission to use this document for purposes other than those enumerated above may be requested by emailing ofmfwg@lists.openfabrics.org. Please include the the requesting individual and/or company and a brief description of the purpose, nature, and scope of the requested use.

DISCLAIMER

Current Revision

The OFA is actively engaged in expanding and refining the Sunfish documentation.

The most current revision can be found on the OFA web site at https://github.com/OpenFabrics/sunfish_docs.

Contact the OpenFabrics Alliance (OFA)

Current OFA practice is to make updates and other information available through the OFA Sunfish github site: https://github.com/OpenFabrics, look for the repositories beginning with "sunfish_".

Requests for interpretation, suggestions for improvement, additions, or defect reports are very welcome. They should be sent via ofmfwg@lists.openfabrics.org.

VERSIONING POLICY

This document is versioned material. Versioned material shall have a three-level revision identifier, comprised of:

1. Version Number: Version numbered material shall be backwards compatible with all of previous revisions of that material that have the same version number. There is no assurance of interoperability or backward compatibility between revisions of a versioned material with different version numbers.
2. Release Number: A release represented by a minor revision number represents a technical change to existing content or an adjustment to the scope of the versioned material. Each minor revision causes the release number to be increased by one. The OFA will make every effort to ensure, but not guarantee, that versioned numbered and release numbered material shall be backwards compatible with previous revisions of the material with the same version number and release number.
3. Errata Number: Contents of an errata revision of versioned material are limited to minor corrections or clarifications of existing versioned material. An errata revision may be backwards incompatible if the incompatibility is deemed necessary for correct operation of implementation(s) of the versioned material.

Future publications of this document are subject to specific constraints on the scope of change that is permissible from one revision to the next and the degree of interoperability and backward compatibility that should be assumed between products designed to this standard. This versioning policy applies to all OFA Sunfish versioned materials.

Revision History

The evolution of this document is summarized in Table 1

Table 1: Revision history

Date	Revision	Notes
Apr 23 2024	0.3	First official release

About the OpenFabrics Alliance

The mission of the OpenFabrics Alliance (OFA) is to accelerate the development and adoption of advanced fabrics for the benefit of the advanced networks ecosystem. The mission is accomplished by; creating opportunities for collaboration among those who develop and deploy such fabrics, incubating and evolving vendor independent open source software for fabrics, and supporting and promoting the use of such fabric technology software.

Acknowledgements

The OFA OFMF Working Group, which developed and reviewed this work in progress, would like to recognize the significant contributions made by the following members listed in Table 2.

Table 2: Contributors

Member Entity	Representative
Intel	Phil Cayton
Sandia National Lab	Mike Aguilar
IBM Research	Christian Pinto
HPE	Russ Herrell
Redhat	Doug Ledford
Intel	Richelle Ahlvers
IBM Research	Michele Gazzetti
HPE	Jeff Hilland
HPE	John Mayfield
IntelliProp	Jim Hull
IntelliProp	Tracy Spitler
Lawrence Livermore National Labs	Chris Morrone
Cornelis Networks	Eugene Novak
Cornelis Networks	Dennis Dalessandro
AMD	Kurt Bowman
Sandia National Labs	Catherine Appleby
Los Alamos National Labs	Alex Lovell-Troy
Fsas Technologies Inc.	Naoki Oguchi
Fsas Technologies Inc.	Jin Hase

Table of Contents

1. Abstract

Sunfish is designed for system administrators, application programmers and users, HPC and cloud architecture designers, and other stakeholders that are involved in the design, deployment, and use of distributed computing systems based on multiple high-speed network fabrics for disaggregating hardware components.

Sunfish provides a universal set of RESTful interfaces, tools and services to manage fabric attached resources, such as CPUs, accelerators, storage, and memory. Sunfish uses the DMTF Redfish Scalable Platforms Management API Specification to allow clients to gather telemetry information on fabrics and components, request information about fabric attachments, allocate components, and compose disaggregated systems. Each vendor specific fabric can be controlled and manipulated through the use of a custom agent that is designed to provide its services and functions to Sunfish via the Redfish API. Sunfish is designed to be versatile and allow clients to connect and interact with underlying high-speed fabrics.

Sunfish provides computing system clients with a common set of tools, to interact with disaggregated fabrics and resources. Clients may include, but are not limited to, Message Passing Interface applications, Fabric Attached Memory (FAM) managers, workload, resource, and cloud managers, IO system managers, storage configuration managers, and CPU and accelerator managers or consumers of such resources. Client embodiments may include, but are not limited to, physical machines, virtual machines, appliances, embedded computational engines, and containers. Any entity (e.g., SW tool, admin GUI, shell script via CLI) with the appropriate permissions may create a virtual platform, pod, cluster, partition, VLAN, job queue, or subnet to enable workload(s) to execute.

2. Document Scope

2.1. Audience Assumptions

As Sunfish is designed as an extension of the Redfish Scalable Platforms Management API Specifica tion, this document is written with the presumption that the reader has a detailed understanding of the Redfish Specification. This document cannot be fully understood without that context.

2.2. Normative References

The documents referenced in Table 3 are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

Table 3: Approved normative references

Title (Version)	Authoring entity	Reference URL
Redfish Specification	DMTF	dmtf.org/dsp/DSP0266
Redfish Data Model Specification	DMTF	dmtf.org/dsp/DSP0268
Swordfish Specification	SNIA	snia.org/forums/smi/swordfish
NVMe Family of Specifications	NVM Express Consortium	nvmexpress.org/specifications/
CXL Specification	CXL Consortium	computeexpresslink.org/cxl-specification/

2.3. Terms and Definitions

In this document, some terms have a specific meaning beyond the normal English meaning. Those terms are defined in this clause including new terms, frequently used Composable Disaggregated Resource Management terms, Sunfish Terms, Redfish terms, etc.

2.3.1. Sunfish-specific Terms

The terms listed in Table 4 are used in this document.

Table 4: Sunfish terms

Term	Definition
Sunfish Agent	The translators between the Sunfish Core Redfish API syntax and schema and the vendor-specific versions used by the given Hardware Manager software.
Sunfish Agent Services	The Redfish API exported by a Sunfish Agent
Sunfish Service	The Redfish API exported by the Sunfish Core

Term	Definition
Actors	Software stack entities or hardware embedded processors
Fabric resources	any resource that might be made available to an application thread as an OS resource; EG. Memory, Fabric Attached Memory (FAM), GPUs, CPUs, storage, remote storage, etc.
Fabric Management Objects	any resource that an administrator might need to manipulate to establish proper behavior of the fabrics that interconnect the composable resources; EG., fabric switches, fabric gateways, firewalls, etc.
Clients	are any of the applications, application libraries (such as libfabric, OpenFAM, or OpenSHMEM), resource managers (such as FAM pool managers, storage pool managers), orchestration managers, workload managers, and the admin GUI and tools that call into the Sunfish CORE's Redfish Services.
Composability Services	a collection of resource managers, policy stores, and monitoring elements for tracking the current state of the entire system (Section 4).
Sunfish Core	The stage of the Sunfish Framework which maintains the aggregate Redfish model of all fabrics it controls and all resources on those fabrics (Section 4).
Hardware Manager	refers to any hardware/software component that is direct control of a set of hardware components (e.g., fabric managers, enclosure managers, BMCs, etc.).

2.3.2. Redfish terms

Many terms in this document were originally defined in the Redfish Specification. Some of the more common terms and definitions are reproduced in Table 5, as an aid to the reader.

Table 5: Redfish terms

Term	Definition
OData	The Open Data Protocol, as defined in the OData documentation
OData Service Document	Resource that provides information about the service root for generic OData clients.
Redfish Schema	Defines Redfish Resources according to OData schema representation. You can directly translate a Redfish Schema to a JSON Schema representation.
Redfish Service	Implementation of the protocols, resources, and functions that deliver the interface that this specification defines and its associated behaviors for one or more managed systems.
Request	A message from a client to a service.
Service Root	Resource that serves as the starting point for locating and accessing the other resources and associated metadata that together make up an instance of a Redfish Service.

2.4. Keywords (normative language terms)

This document conforms to ISO/IEC Directives, Part 2 for keyword usage. The most common terms and their intended meanings are summarized Table 6.

Table 6: Normative language terms

Term(s)	Meaning
shall / shall not	Used to identify objectively verifiable criteria to be fulfilled and from which no deviation is permitted if compliance with the document is to be claimed
should / should not	Used to identify a suggested possible choice or course of action deemed to be particularly suitable without necessarily mentioning or excluding others
may / need not	Used to convey consent or liberty (or opportunity) to do something
can / cannot	Expected or conceivable material, physical or causal outcome
must	Identifies a constraint or obligation on the user of the document, typically due to one or more legal requirements or laws of nature, that is not stated as a provision of the standard *NB:* "must" is not an alternative for "shall", and should only be used for constraints that arise from outside this standard

3. Sunfish Framework

3.1. Introduction

Sunfish is a centralized management platform for Composable Disaggregated Infrastructure (CDI). Sunfish provides a method by which compute resources, that are attached by high speed, low latency networks (e.g., CXL/PCIe, RDMA, Ethernet), can be assembled together and managed as if the components were part of a traditional compute server.

This chapter provides a description of Sunfish and attached components that provide organized assembly of material resources and managers that provide the hardware associations used in composing systems out of disaggregated components.

3.1.1. Overview of Composable Disaggregated Infrastructure

With Composable Disaggregated Infrastructures (CDI), computational resources are not statically provisioned in servers, but instead are physically disaggregated in shared pools and connected through high-speed/low-latency network fabrics. These resources may be dynamically provisioned and re-provisioned to client applications, as needed, and are thus not only more efficient to manage by removing unnecessary hardware, but help reduce energy consumption and datacenter cooling costs. CDI enables assigning of pools of resources to consumers. Remote resource disaggregation is already common for storage devices (e.g., NVMe-oF); current trends are pushing this paradigm further, extending it to assigning private or shared resources such as computational engines, disaggregated memory elements, disaggregated accelerators, and eventually to all forms of compute resources required by modern HPC applications.

CDI eases issues currently plaguing today's HPC, Cloud, AI, etc. architectures. In current large-scale compute architectures, clusters are created by combining separate servers over shared network fabrics. Traditional compute servers in clusters are statically provisioned and assembled with their own CPUs, memory devices, accelerator cards, and storage devices contained within a fixed size server enclosure. The need to incorporate "all of the options that may be required to support a given workload" often results in resource over-provisioning, limits parallel workloads, makes traditional HPC architectures less flexible and less efficient, and can lead to situations where application jobs are more prone to run-time failure. Resource over-provisioning and inefficient use of hardware are common issues to any large scale computing facility.

3.1.2. Managing Composable Disaggregated Infrastructure

The management and optimization of such a diverse set of fabrics and fabric technologies to realize the benefits of Composable Disaggregated Infrastructures is quickly becoming a complex issue to solve for infrastructure managers, especially in heterogeneous multi-vendor environments, with multiple vendor-sourced hardware and the ever-expanding collection of proprietary APIs and tools.

The larger the HPC or Cloud system, the greater the potential impact of dynamic CDI to improve both energy efficiency and computational stability. Composability Services are needed to efficiently connect workloads with resources in a dynamic ecosystem, at scale, without concern for the management model of the underlying hardware technology. Centralized composability management must be scalable to be able to handle massive amounts of hardware telemetry, device states, device capabilities, and subscription information from large numbers of resources.

Using centrally managed CDI on large-scale resources can lead to more efficient distribution of the resources, increased batch run performance, and allow measurable performance where efficient resource scheduling is as important as physical size. As such, integration of newer HPC Workload Managers and Container deployment systems should be part of a master plan for centralized CDI management. With aggregation of resources to meet parallel batch run requirements, CDI aggregation of resources should be made only after full verification that those aggregations can be made successfully.

3.1.3. Sunfish/CDI Use Case Examples

Sunfish and its components provide scalable central CDI management for large-scale, heterogeneous computing systems. The core component of Sunfish, the redfish database, acts as the 'source of truth' for both aggregation and resource attachment for the current state of the running system. Sunfish Agents provide real-time interaction with hardware and network management components and are tasked with providing 'real world' hardware configuration and provisioning. A Composability Service allows for complex integration with both workload and container deployments. The Composability Service attempts to associate client workloads with requested resource components.

3.1.3.1. More Efficient Sharing of Resources Through Resource Pools

In current non-CDI HPC systems, hardware resources necessary to support every type of computation must be installed into each compute node. Batch job execution is generally managed using a workload manager (e.g., Flux) or through container deployment (e.g., Kubernetes) and each batch job occupies a fixed subset of compute nodes. This type of system architecture leads to stranded and wasted resources and limits the resources which may be available to other user jobs. For instance, if a user job requires six GPUs, 1 TB of on-board memory, and eight CPU cores in each node to complete a batch job, then the job cannot be run on a cluster with four GPUs and four CPU cores. In addition, if a batch job needs only two GPUs and two CPU cores, the other two GPUs and six CPU cores are wasted resources that can't be allocated to other batch jobs.

Some workload managers or container deployment systems are very capable of time-sharing many of the multi-tasking resources within their control and thereby optimizing the utilization of those resources during the execution of the job streams on cluster sizes for which they are designed. Bringing multiples of such workload managers or deployment systems into a composable HPC environment then requires a methodology for composing smaller, virtual clusters which can be handed over to these 'virtual cluster managers' which are tailored to specific classes of workloads. In large HPC installations, there can be hundreds or thousands of virtual cluster managers watching over their private, composable resource pools, so it is essential the Sunfish Composability Services are architected to be hierarchically scalable.

3.1.3.2. Sharing of Memory resources

The CXL 3.1 specification enables remote memory to be pooled or shared. Figure 1 shows a simple example of memory sharing. On the left side, the orange node is connected to the orange NVMe memory through a CXL switch. On the right side, the CXL components have been configured to allow the orange node to take a smaller portion of a shared NVMe memory. The blue node, needing more memory resources, has a portion of the shared memory and another CXL memory, again connected via a CXL fabric switch. Each of these CXL devices and switches potentially have their own Hardware Manager entities, each of which must be informed of the required configuration details. The Sunfish Framework provides administrators a single API through which to manage all these different components.
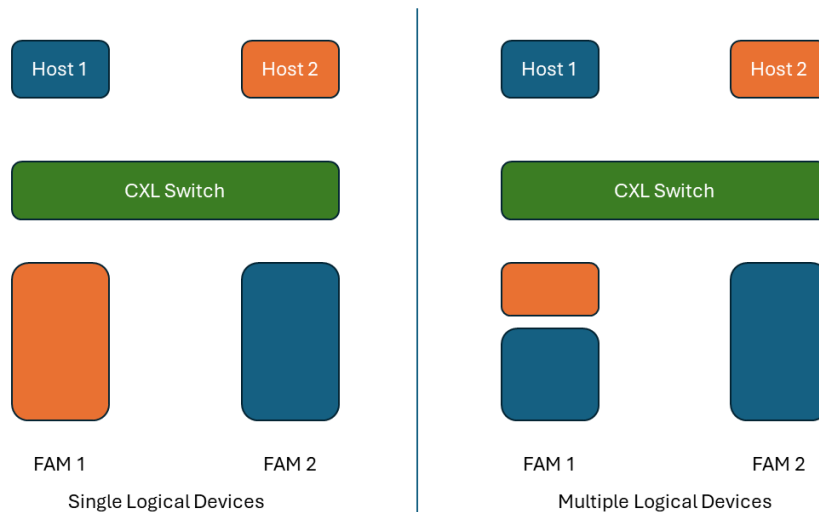
Figure 1: Sharing of CXL FAM Resources

3.1.3.3. Composable Accelerators

The cost of acquiring, maintaining, sharing, and cooling computational accelerators has begun to be an issue for large-scale parallel systems, such as HPC and Cloud computing platforms. CDI, through the use of fabrics and CXL can help provide accelerator resources, with peer-to-peer communication capabilities, when and where they are needed in batch computations.

As an example, GPUs can be managed by Sunfish as network fabric attached pools, or as CXL fabric pools, as part of a peer-to-peer resource sharing architectural arrangement, across compute nodes. The Sunfish Core database provides entry links that reflect the current attachment arrangement.

3.2. Sunfish Framework Scope

3.2.1. Goal

Provide clients of Sunfish with a vendor and fabric agnostic API which enables them to view manage and orchestrate the resources in a CDI scenario. Users will be able to query all the resources in the available fabrics, understand their status, manipulate their state and finally compose them into virtualized compute nodes that can then be assigned to workloads directly or via virtual machines and containers.

3.2.2. Strategy

The OFA Sunfish team is using the following strategy to define the Sunfish Framework:

- Standardize on models of composable fabric resources based on the DMTF Redfish and SNIA Swordfish schema.
- Analyze specific tasks which applications and administrators need to perform to access and/or manage composable fabric resources (and 'fabric management objects') at an appropriate level of abstraction. Specific tasks are use cases (e.g., creating shared regions of FAM, binding/mapping shared regions of fabric attached memory to specific CPUs (hosts), placing specific fabric resources under control of specific orchestration tools, etc).

- Extract a suitable model of such abstracted resource objects and functional operations, and define the necessary actors and their roles in the message and work flows (APIs) that accomplish the specific tasks.

3.2.3. Deliverables in This Document

This document describes the approach and architecture of the sunfish framework, its software and the API that actors use for interacting with a Sunfish service. This document is also defining the *Sunfish compliant* interpretations of Redfish and Swordfish schemas. All APIs and interactions are based on DMTF Redfish and SNIA Swordfish specifications. New schema objects are introduced and detailed whenever necessary.

This document is a living document, and readers should expect frequent updates, additions and corrections as the various problems to be solved are addressed and as the various related standards evolve.

4. Sunfish Framework

Sunfish is designed to configure fabric interconnects and manage composable, disaggregated resources in dynamic highly distributed compute infrastructures using client-friendly abstractions. Sunfish provides a framework for abstraction of, and communication with, the multitude of independent management tools behind a single, consistent, standards-based API; it does this through a universal set of RESTful interfaces and tools and services to manage fabric attached resources, such as, CPUs, Accelerators, and Memory Devices. Sunfish uses the common languages of Redfish and Swordfish, to allow clients to gather telemetry information on fabrics and components, request information about fabric attachments, allocate components, and compose disaggregated systems. Each vendor specific fabric can be controlled and manipulated through the use of a custom agent that is designed to provide its services and functions to Sunfish via the Redfish API. Figure 2 presents the concepts of the Sunfish Framework in visual form.
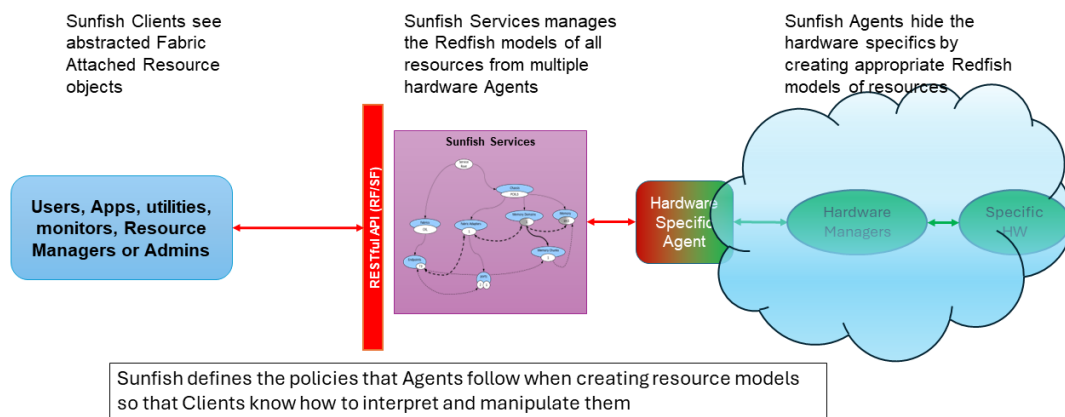
Figure 2: Sunfish framework objectives

Figure 2 makes several important points:

- On the right we have our disaggregated resources with potentially many different management entities in direct control using vendor and device specific means.
- We need an Agent layer (aka 'provider') to aggregate the inventory from one or more Hardware Managers and convert the hardware specific device and resource descriptions into a publicly accepted and commonly interpreted Redfish model.
- The Agent hands this whole model over to the Sunfish services. This effectively abstracts much of the hardware specific details from the clients.
- Clients of Sunfish services see the Redfish models of resources and manipulate these models to establish or alter the state of individual resources or their assignments (bindings) to consumers (e.g., hosts).
- As it is critical that Agents and Clients have the same interpretation of a Redfish object found in the Sunfish database, the Sunfish framework also has policies and requirements to be followed when creating or interpreting the Redfish models.

4.1. Components of the Sunfish Framework

Figure 3 depicts the major software components (layers) that make up the architecture of the Sunfish Open Fabric Management Framework:
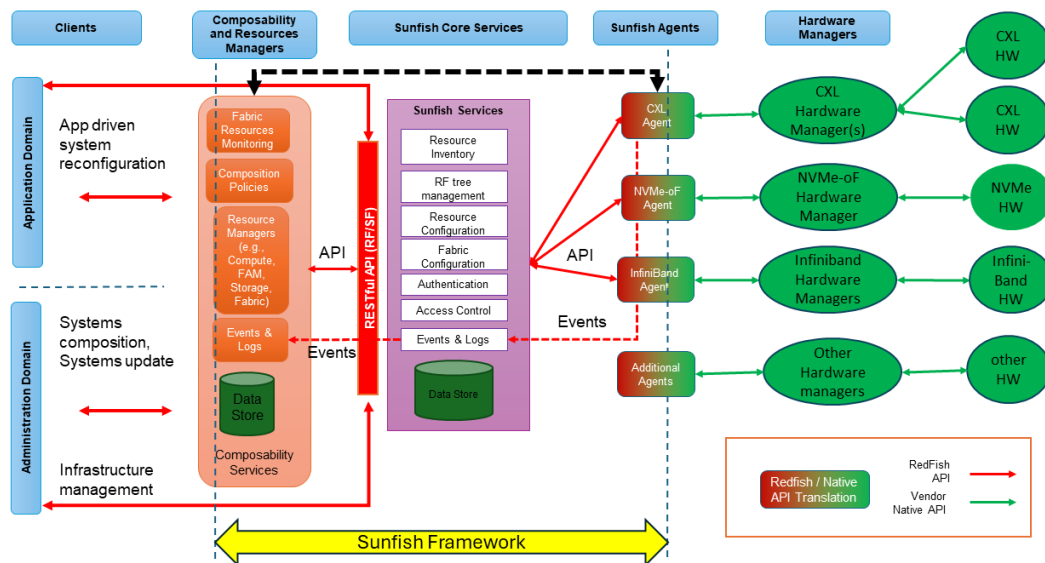
Figure 3: Sunfish framework components

CDI requires some modest amount of scale, which in turn likely requires a Redfish representation of many different resource types residing on different interconnects, all supplied by different vendors. Disaggregated resources and interconnect fabrics may or may not support a Redfish API suitable for directly plugging into the Sunfish management framework, so the presence of an API translating Agent entity is used in all cases, though the complexity of any given Agent obviously depends upon the actual fabric management APIs and policies in use in the *Hardware Manager's* realm. Sunfish not only collects and manages the Redfish object models for disaggregated resources and the fabrics that connects them, but also serves up control points for many common management tasks, such as Events and Logs, Access Control settings and policies, and authentication needs.

4.1.1. Clients

Clients are any of the applications, application libraries (e.g., libfabric, OpenFAM, or OpenSHMEM), resource managers (such as FAM pool managers, storage pool managers), orchestration managers, workload managers, and the admin GUI and tools that call into the Sunfish CORE's Redfish Services.

Client software queries the Sunfish Core Redfish service (e.g., using HTTP GET) to determine what resources are available on the fabric. Clients issue updates (e.g., using HTTP PUT, POST, PATCH, DELETE) to objects to make modifications to the Redfish fabric model.

Clients identify the fabric resources by the URIs given them by the Sunfish Core's Redfish service. When clients need to exchange information about fabric resources, they use the Sunfish Core's URIs to unambiguously refer to specific objects.

Below is an example Sunfish URI:

```
/redfish/v1/Systems/1
```

Clients accept the Redfish object description as the current state. The URIs in use within the Sunfish Core Redfish service are unique only to that Service and make up that Sunfish Core's Redfish URI namespace.

4.1.2. Composability Services

The element that sits between clients and the Sunfish Core block is collectively called the Composability Services. The Composability Services can be seen as a collection of resource managers, policy stores, and monitoring elements for tracking the current state of the entire system in relation to the dynamic composition of disaggregated components. The Composability Services is in charge of composing hardware resources according to requirements coming from Clients and higher-level, possibly proprietary composition and orchestrator tools. Technically, the components of the Composability Services are all clients of the Sunfish Core Services and any user or admin application can pursue taking responsibility for these tasks. However, within the Sunfish Core architecture we anticipate releasing a coordinated set of reference code utilities to offer random clients commonly needed 'composability' features.

Examples of such features include, but may not be limited to:

- Composition/Deletion of composed computer systems and virtual clusters
- Dynamic update of computer systems resources
- Application driven dynamic resources configuration and provisioning such as:
  - Creation and binding of memory and storage resources
  - Creation and binding of shared memory region across nodes
  - On-demand provisioning of accelerators
  - Virtual network creation
- Management of client permissions/resource ownership

Coordinating these various clients enforces a cohesive interpretation of the Redfish objects and schema which are used to describe the fabric resources being managed. In addition, permissions and access controls for shared fabric resources need to be managed in this layer, which is above the Sunfish Core which enforces access controls and below the general Clients upon which the restrictions are imposed.

4.1.3. Sunfish Core Services

The Sunfish Core maintains the aggregate Redfish model of all fabrics it controls and all resources on those fabrics. The Sunfish Core Services are presented to all clients via Redfish API calls into the Sunfish Core library. When clients request data or request changes to model objects, the Sunfish Core determines which Redfish objects in the model are impacted, makes the required changes to those Redfish objects in the model. Any relevant actions or requests that affect state or configuration of the fabric manager or actual fabric hardware are relayed by the Sunfish Core to the fabric-specific Sunfish Agent.

The Sunfish Core also subscribes to events from the various Agents and offers its own Events Subscription Service to Sunfish Core clients. Clients and the Composability Service can subscribe to the various Redfish events defined for the Redfish objects, and when the Sunfish Core receives associated events from the Agents these events are forwarded to those subscribers.

Not all client requests will require the Sunfish Core to interact with a fabric Agent. Many requests (e.g., a request to register to receive certain events) simply affect the Sunfish Core fabric model and some bookkeeping properties therein. Some simple client requests (e.g., to DELETE an object) may translate to multiple changes to multiple model objects and potentially require multiple exchanges with the fabric Agent.

Maintaining the integrity and consistency of the aggregate Redfish model of all fabric resources is one of the primary duties and major values of the Sunfish Core and associated Agents.

Finally, the Sunfish Core is responsible for tracking and enforcing Authentication and Access Control policies for both Clients and Agents.

4.1.4. Sunfish Agents

Hardware-specific Sunfish Agents act as the translators between the Sunfish Core Redfish API syntax and schema and the vendor-specific versions used by the given Hardware Manager software. Agents thus expose a Redfish interface to the Sunfish Core, and use a (potentially) fabric-specific protocols to an actual Hardware Manager. E.g., the Hardware Manager may have a RESTful interface called 'bind resource' which allows an admin to enable Host A to access Memory B. The Sunfish Core's Redfish equivalent is 'POST Connection' between Host A and Memory B.

Fabric-specific agents also act as the translator between the Sunfish Core Redfish URI namespace and the fabric specific component and resource namespaces. E.g., the open source Gen-Z fabric manager for Linux (called Zephyr) assigns its own 128-bit UUID-style ID to a Gen-Z fabric memory module. Redfish models a complicated fabric resident memory module as several related Redfish objects (Fabric Adapters, fabric Ports, fabric Endpoints, Memory Domains, etc). The Gen-Z agent is responsible for keeping the mappings between the various Redfish IDs (URIs) assigned by the Sunfish Core and the associated IDs (UUIDs) assigned by the Zephyr Fabric Manager. Clients use the Redfish IDs, and the Hardware Manager uses the Hardware Manager IDs. The agent is possibly the only entity that knows both namespaces.

Like the Sunfish Core, an agent needs to parse requests coming from its client (the Sunfish Core), modify its internal representation of its fabric view, update any internal state it is required to track, and send any appropriate request or requests on to the appropriate fabric manager.

Another very important role of the agent is to aggregate the inventory and events from multiple Hardware Manager instances used to configure and control larger fabrics as Figure 3 illustrates. Since a large fabric may have multiple Hardware Managers controlling components on the fabric which must share a common fabric address namespace, **the task of reconciling multiple namespaces and 'subnets' into a single Redfish Fabric representation falls to the agent.** Even when the Hardware Managers' APIs are also Redfish, the agent is responsible for sequencing appropriate Redfish commands and/or actions to the appropriate FMs and coordinating the multiple returns into a suitable response to the Sunfish Core.

Finally, the agent passes events, alerts, and performance monitoring updates in support of the Sunfish CORE's Redfish events service. The Agent – Hardware Manager interface is Hardware Manager specific. The Sunfish Core – Agent interface is architected to be a Redfish API using Redfish schema, so the agent must translate events and logs emitted by or retrieved from the FM into the appropriate Redfish formats.

4.1.4.1. Agents Representation in the Sunfish Model

The Sunfish Core component maintains the complete state of the system it oversees, in the form of a Redfish tree. Therefore, each object managed through Sunfish (e.g., memory, server, switch) has its own Redfish counterpart which will consist of one or more Redfish object(s). Agents are represented in Sunfish by means of the `AggregationSource` object that in Redfish represents the source of information for the resources it *aggregates*.

The below snippet shows an example *AggregationSource* for a fictional CXL Fabric agent. The `HostName` field contains the endpoint for connecting to the specific agent. The language to be used for interacting with agents is specified in the `ConnectionMethod` field contained in the `Links` section of the object. There are multiple possible options for a connection method, but Sunfish will always use **Redfish** for connecting with its agents. In Sunfish, we keep the connection method field only for the sake of being complete with respect to the Redfish specification.

```
{
    "@odata.id": "/redfish/v1/AggregationService/AggregationSources/UUID",
    "@odata.type": "#AggregationSource.v1_2.AggregationSource",
    "HostName": "http://hostname:port",
    "Id": "UUID",
    "Links": {
        "ConnectionMethod": {
            "@odata.id":
"/redfish/v1/AggregationService/ConnectionMethods/CXL"
        },
        "ResourcesAccessed": [
            "/redfish/v1/Fabrics/CXL"
        ]
    },
    "Name": "Agent UUID"
}
```

An example `ConnectionMethod` object is shown below.

```
{
    "@odata.type": "#ConnectionMethod.v1_1_0.ConnectionMethod",
    "Id": "CXL",
    "Name": "ConnectionMethod for Agent CXL",
    "ConnectionMethodType": "Redfish",
    "@odata.id": "/redfish/v1/AggregationService/ConnectionMethods/CXL"
}
```

The `ConnectionMethodType` must be set to `Redfish` for all Sunfish Agents.

Each agent is assigned a UUID upon the registration with Sunfish, when the associated `AggregationSource` object is created. Sunfish returns the UUID to the agent by patching the Agent's internal state. More details on the UUID generation are provided in Section 4.3.1. All implementations of Sunfish shall expose the `AggregationService` in their main Redfish tree.

4.2. Hardware Managers

The term 'Hardware Manager' refers to any hardware/software component (e.g., a device manager, an enclosure manager, a BMC) that is in direct control of a set of hardware components. The Gen-Z and CXL specifications use the term 'fabric manager (or FM) to indicate this directly controlling manager. Since the

terms 'fabric manager' and 'fabric management' carry many different interpretations in many different contexts throughout the industry, the Sunfish Framework will avoid these terms.

Thus, for the purposes of the Sunfish Core Architecture, Hardware Managers are those entities with physical access to the control space of the fabric resources and the authority to modify those settings. A Hardware Manager is responsible for performing a fabric crawl, taking inventory of fabric resources, and the initial configuration of such resources as required by the Hardware Manager's initial configuration policies. The Hardware Manager's 'management domain' is the set of those components for which it has primary access rights to the associated control surfaces. Managing hardware components usually requires vendor-specific knowledge, so the Sunfish Framework does not offer reference Hardware Managers; these are left to vendors to implement.

4.3. Sunfish Framework Events

Agents are resource aggregators that present Sunfish with a Redfish / Swordfish model of all resources which they manage and/or model. Sunfish-agent communications are event-driven. Events are exchanged upon every change in the state of the overall system, be it a new component being added, a component reporting a failure, etc. The format of the events exchanged between Sunfish and clients/agents is defined as part of the Redfish schema. Redfish defines an Event Message Registry [1] that dictates the format and type of all legal Redfish events.

Redfish events are based on the subscription model, where a Redfish API instance will generate events only towards subscribers based on the characteristic of the event to be sent, and the filters specified in the available subscriptions. Subscriptions are used by Sunfish clients to register to events of interest, as well as the Sunfish Core itself to register to all events generated by all hardware agents. Subscriptions are represented as `EventDestination` Redfish objects part of the EventService exposed by the Redfish endpoint generating the events.

4.4. Interactions Between Sunfish and Hardware Agents

4.4.1. Hardware Agent Registration

Each hardware agent must first register with the Sunfish Core service before its resources can be integrated and managed from the Sunfish main endpoint. The registration and discovery of agents is performed through a handshake triggered, usually, at agent startup. All the interactions part of the discovery handshake are based on Redfish events (Section 4.2). The handshake process is depicted in Figure 4. As explained in Section 4.2, events in Redfish are based on the subscription model, where an `EventDestination` object exists for each subscribed entity. Each Sunfish Agent exposes an Event Service with a single subscription pre-created for the Sunfish Core Event Service. In this way any event generated by the agent is always sent to Sunfish. The Sunfish specification does not prevent other actors from subscribing to agent events. An `EventDestination` should be populated preferably prior to the agent startup, and necessarily before an event is emitted by the agent. The information contained in such object are the URI and port for reaching the Sunfish Event Service as well information for encryption of messages (i.e., certificates).
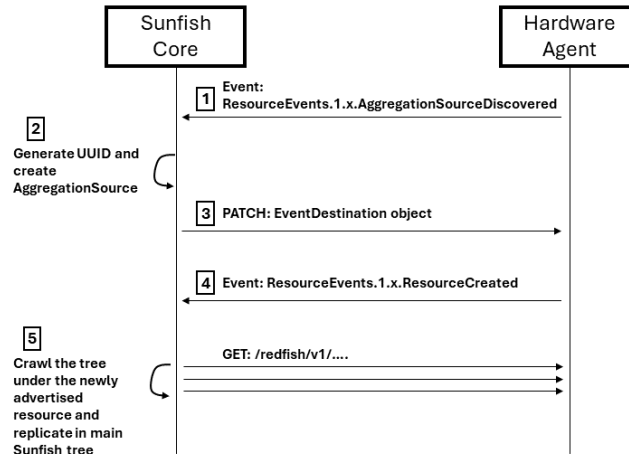
Figure 4: Sunfish-Agent handshake process

The handshake process is depicted in Figure 4. The registration process starts (1) with the agent sending an `AggregationSourceDiscovered` event to the Sunfish *EventListener* service. Such event carries the connection method type, *Redfish* for any Sunfish applications, and the service endpoint in the format `hostname:port`. Information is passed through events by mean of the `MessageArgs` field with their format being mandated by the Redfish Message Registry[1]. An example event is shown in the below snippet.

```
{
    "@odata.type": "#Event.v1_7_0.Event",
    "Name": "AggregationSourceDiscovered",
    "Context": "",
    "Events": [ {
        "Severity": "Ok",
        "Message": "A aggregation source of connection method
                    Redfish located at http://hostname:port
                    has been discovered.",
        "MessageId": "ReosurceEvent.1.x.AggregationSourceDiscovered",
        "MessageArgs": [ "Redfish", "http://hostname:port" ],
        "OriginOfCondition": {
            "@odata.id":
"/redfish/v1/AggregationService/ConnectionMethods/NAME"
        }
    } ]
}
```

Once Sunfish receives the event, it will (2) generate a 128-bit UUID associated to the registering agent and create an `AggregationSource` as described in Section 4.1.4.1. Once the registration is completed, the Sunfish Core is sending the UUID back to the agent by patching (3) the `Context` field in the event subscription that the agent has pre-populated with the Sunfish Core connection details.

Once the agent is registered, it will send one or more events to advertise those resources that are to be managed through Sunfish. After the registration is successful, any further event sent by an agent will contain the UUID of the registered agent in the `Context` field of the event payload. Each agent can retrieve the UUID from the `Context` field in the EventDestination that keeps the information for sending events to the Sunfish Core. For each resource to be advertised, a `ResourceCreated` event is sent containing the new resource URI in the `OriginOfCondition` field in th event json description. For each of these events received, Sunfish will crawl the tree that has the resource as root, and add all the resources in its own global view of the system. While crawling, for each further resource to be visited, a GET operation is issued to the agent to get all the details on the resource itself. Once added to the tree, the resource is also added to the `ResourcesAccessed` field in the `Links` section of the `AggregationSource` object associated to the agent.

```
{
    "@odata.type": "#Event.v1_7_0.Event",
    "Name": "Fabric Created",
    "Context": "Agent: UUID",
    "Events": [ {
        "Severity": "Ok",
        "Message": "New Fabric Created ",
        "MessageId": "ResourceEvent.1.x.ResourceCreated",
        "MessageArgs": [],
        "OriginOfCondition": {
            "@odata.id": "/redfish/v1/Fabrics/CXL"
        }
    }
    ]}
```

As a result of the crawling process all resources part of the sub-tree of the advertised resources are brought into the Sunfish tree. At the same time, all resources discovered through links are also added to the Sunfish tree, even if they do not directly belong to the sub-tree of the advertised resource.

4.4.2. Forwarding Requests to Hardware Agents

To facilitate associating a resource in the Sunfish Redfish tree with the managing agent, all resources are marked during the crawling process (Section 4.3.1) with the agent UUID. Objects are marked using the `Oem` field that, according to the Redfish specification, can be used by organizations for adding custom fields to Redfish objects. The below snippet shows the structure of the Sunfish Oem extensions used for marking objects. The `ManagingAgent` field contains the full Sunfish URI of the managing agent `AggregationSource`. This structure helps the Sunfish internal services to quickly associate an agent to an object while performing operations on the Redfish tree. An example is a fabric connection being created. In order for the actual connection to take place, the request should be forwarded to the agent managing the specific fabric. The URI contained in the Oem extension field of the Fabric object would identify the managing agent, without the need for looking up across all the available agents.

```
{
    "Oem": {
        "Sunfish_RM":{
```

```
            "@odata.type": "#SunfishExtensions.v1_0_0.ResourceExtensions",
            "ManagingAgent": {
                "@odata.id":
"/redfish/v1/AggregationService/AggregationSource/AgentID"
            }
        }
    }
}
```

4.5. Failover / Failure-recovery

The following scenarios are example failure and restart situations that the Sunfish architecture is intended
to cover:

- Agent Restart: A running fabric has an up-to-date Sunfish Core Services model of the fabric,
  provided by a specific instance of an Agent which, in turn, is in communication with one or more
  running Hardware Managers. Then the Agent software instance encounters a fault and restarts. The
  Sunfish Core Services' Client APIs and services may be momentarily stalled, but are not terminated.
  The fabric Hardware Managers remain functional with no service interruptions to completely
  configured virtual platforms. Agent services are unavailable until the reboot completes and the Agent
  software rebuilds its state and internal fabric models from the Hardware Managers' state and the
  alerts Sunfish Core that it must re-load the fabric model in case anything changed on the hardware
  while the Agent was offline.
- Agent Failover: The same behavior applies to the Sunfish Core Services and the fabric hardware
  functionality, but when the Agent software instance encounters a fault it shuts down rather than
  reboots. A secondary (failover) Agent detects the failure of the primary Agent, securely terminates
  the primary Agent and assumes the role of the primary Agent in a fashion similar to the Agent Restart
  scenario.
- Hardware Manager restart or failover: A running fabric has an up-to-date Sunfish Core Services
  model of the fabric, provided by a specific instance of an Agent which, in turn, is in communication
  with one or more running Hardware Managers. If any of the Hardware Managers encounters a fault or
  otherwise determines that its internal state may be inconsistent with the hardware, the Hardware
  Manager restarts or failovers to a secondary Hardware Manager without disturbing the current state
  of the fabric hardware.

The primary axioms in all the above scenarios are:

1. Any restart of the software above the fabric hardware **shall not** disturb the current state of the fabric
   hardware in the process of rebuilding software's state.
2. The current state of the fabric hardware is the correct state that should be reflected in the Sunfish
   Core. All Hardware Managers, Agents, and the Sunfish Core **shall** rebuild their internal models to be
   consistent with the current fabric hardware state.
3. Sunfish Core Service restarts or failovers require that the Sunfish Core Service **shall have** a
   mechanism to 're-discover' all the Agents that are currently still running.
4. Sunfish Agents likewise **shall have** a mechanism to re-discover any existing Hardware Managers
   running.

To discern whether the current state of a Core Service, an Agent instance or a Hardware Manager is consistent with current running hardware the Sunfish Framework proposes some basic requirements for Hardware Managers and Agents to supply the equivalent of UUIDs to the Sunfish Framework for certain basic

- Hardware Managers that feed the Agent the inventory and status of the fabric components they are managing **shall** apply some form of Hardware Manager created UUID that identify the Hardware Manager instance that created the inventory. The Agent **shall** include a form of that UUID in a Fabric ID that the Agent supplies when it registers with Sunfish Core and sends the aggregated Redfish model for the resources reported by the Agent. This ties a Sunfish fabric model to a set of specific instances of Hardware Managers managing the associated hardware.
- Each individual resource reported by the Hardware Manager to the Agent **shall** have a unique UUID that is fixed to the hardware components.
- An Agent that restarts can retrieve the current state of the Hardware Managers' resources. If the Hardware Managers have not reset, each will return the same Hardware Manager instance UUID to the restarted Agent. When the Agent re-uses those same UUIDs to generate a Fabric ID, it **shall** register with the same Fabric ID as last time.
- When the restarted Agent attempts to register the same Fabric ID and an inventory that contains (mostly) already present hardware UUIDs, this is a major tip-off to Sunfish Core that this 'new Agent' is managing an existing fabric and not a totally new version.
- If it had been the Hardware Managers that restarted and not the Agent, the Hardware Manager instance UUID will change and the Agent can detect the mismatch when the Hardware Manager registers with the Agent. Since the Agent already has an inventory containing most or all of the same hardware UUIDs, the Agent can map the new Hardware Manager UUID to the original Hardware Manager UUID that the Agent and Sunfish Core are going to continue to use as a handle for communications involving these resources.
- When the Hardware Manager restarts, it **shall** re-discover the fabric components in its management domain. It is pretty obvious if the Hardware Manager is crawling out a running fabric (components in 'managed' states) vs a cold fabric (un-managed components). The Hardware Manager **shall** alert the Agent that it is reporting a 'cold start' fabric inventory.
- The Agent and/or Sunfish Core **shall** have checks in their Hardware Manager and Agent registration process (respectively) to detect an upload of an existing fabric inventory and status, and handle both scenarios unambiguously.
- If the hardware of the fabric was not reset, the Hardware Manager and the Agent **shall** transfer the current fabric inventory and state to Sunfish Core, preferably as a series of ResourceUpdated Events that eventually bring the resource models and inventory for all three entities (Sunfish Core, Agent, Hardware Manager) into alignment with the hardware state. Sunfish Core **shall** issue the appropriate update Events to all correspondingly registered clients.

It becomes necessary for the restarting or newly attaching Hardware Manager or Agent to locate that next entity in the Sunfish Framework, re-register with it, and notify it of new or possibly updated inventory and state is available. The Agent or Sunfish Core Service is responsible for detecting the possible scenarios:

- New Hardware Manager reporting existing inventory (e.g., Hardware Manager restart)
- Old Hardware Manager reporting inventory change (e.g., rebooted hardware)
- New Agent reporting existing inventory (e.g., Agent restart)

The basic process is the same for any of the detected restarts or fail overs: Sunfish Core eventually learns of new or updated inventory and re-fetches the entire fabric inventory. However, Sunfish Core Services **shall** cross reference the UUIDs of all fetched Redfish objects with all UUIDs of the existing Sunfish Core Redfish tree looking for duplicate UUIDs with different Redfish URIs and resolve the disconnects.

Unfortunately, when Hardware Managers restart or failover they may not have access to state that is created by the Sunfish Core or the Agents. Not all Redfish models of 'logical' resources are forwarded to the Hardware Managers, even if the Hardware Managers support a Redfish API for their Agents / Clients to use. If Hardware Manager state about these 'logical' resources is lost when the Hardware Manager restarts or fails over, the hardware itself may not have these details in a recognized format when hardware state is uploaded again by the Hardware Manager. The hardware state is the end result of a sequence of configuration requests, and the intents of the individual requests may be lost. E.g., the Hardware Manager can read the routing tables of a fabric switch and can determine which fabric paths exist between which physical ports on the fabric. However, the Hardware Manager cannot reconstruct the possibly many Redfish Zone objects that were POSTed by the Sunfish Core Services that led to these particular switch routing table contents.

The preceding problem also applies to Agents that restart or fail over; Agents may also be missing models of 'logical' resources created by the Core Service that are not present in the fresh inventory the Agent obtains from the Hardware Manager.

Reloading the hardware state to the Sunfish Core Services will give Sunfish Core, the Agent(s), and the Hardware Manager(s) the same consistent view of the hardware state. Then, Sunfish Core Services **shall** determine the set of Redfish objects not present in the uploaded inventory that are 'logical' resources that were defined above the Hardware Manager(s) and/or Agents. Sunfish Core Services **shall** then POST or Patch the Agents' Redfish models to make them consistent with the 'logical' view of the fabric hardware. To properly rebuild the model of the fabric across all three entities that are tracking it (Core Services, Agent and Hardware Manager) requires information flow in both directions along the Sunfish Framework. Obviously, if more than one entity fails, there may be no way to recover all the details properly.

If Sunfish Core Services restarts, there may be considerable Client supplied Redfish objects that are not recoverable from the Agent(s). However, state from within the Agent will be sufficient to restart Sunfish Core Services with an accurate view of the current fabric state. Some Client supplied Redfish objects have to be recovered from the Clients themselves.

4.6. Sunfish Access Rights and Permissions

Access rights to Sunfish Core Services will use the standard Redfish rights and permissions capabilities to establish which Redfish objects are visible and modifiable by which Clients of Sunfish Core. Also Sunfish Composability Services (and the utilities therein) will supply additional access filtering. For example, all accesses to a CXL FAM resource's management structures may be req

4.7. Sunfish Interpretation of the Redfish Fabric Model

Figure 5 depicts a simplified Redfish model of a CXL fabric that has two switch hops between a host CPU and a Fabric Attached Memory (FAM) controller.
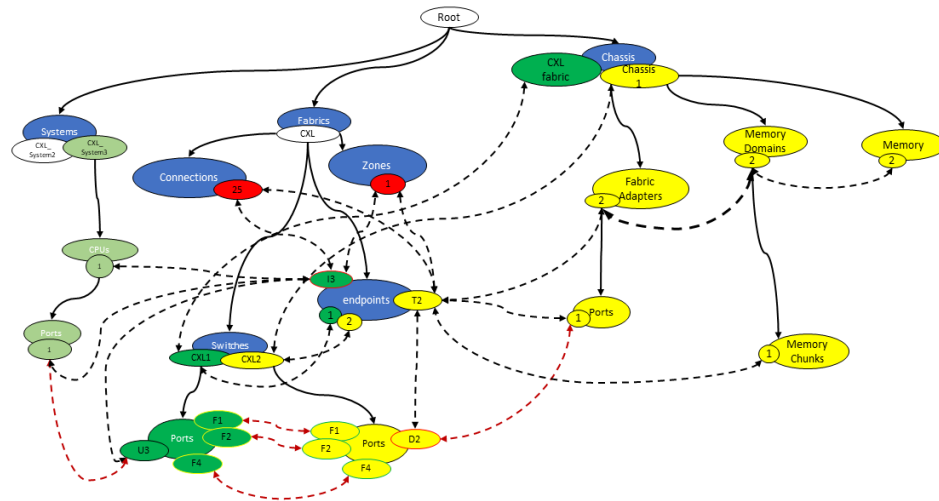
Figure 5: Redfish Model of 2 Switch CXL Topology

The fabric itself (i.e., the physical connectivity) is modelled as **Ports** and **Endpoints**. Physical fabric connections (e.g., cables) are always shown as between 'Ports'. Ports are traced back to the fabric devices that drive the ports. Fabric devices trace back to physical or logical infrastructure which controls the fabric device and to the 'Endpoint' object associated with the fabric device. In the diagram above, the physical fabric links have been highlighted as red dotted links.

The host is modeled as a CXL compute **System** with a **CPU** that has a CXL port linked to the fabric **Switch** CXL1. The FAM controller is modeled as a **Fabric Adapter** that has a CXL port connected to the fabric at Switch CXL2. The host is an Endpoint on the fabric, as is the Fabric Adapter. The model above shows the two Switches have associated Endpoints, but that is often not the case. Switches are only modeled as Endpoints on the fabric if they source or sink fabric traffic. Most switches will only source and sink management plane traffic and not data plane traffic so they often have no Endpoints associated because the Redfish model is of the data plane.

An Endpoint can be viewed as a destination, such as a server connected by a network card or a switch port. A resource can be considered as a component such as a FAM memory controller that provides services or resources to a fabric. The physical topology dictates which resources (Endpoints) have a physical path over which they might communicate. The definition of a *switched* fabric implies all Endpoints of the fabric have physical paths over which they might communicate. There usually is a way to limit which Endpoints may communicate with which other Endpoints. The actual mechanism used is fabric-specific, as in vLans on Ethernet. In Redfish fabrics, which physical paths are enabled between two Endpoints is specified by listing those Endpoints in a **Zone.**

A Zone can be viewed as a set of Endpoints that are allowed to pass traffic among the Zone members, such as the compute nodes and FAM nodes of a compute cluster. Zone descriptions can be used to determine the switches' route table contents to enable traffic between Zone members. A **zone-of-zones** can be considered to be a collection or union of Zones.

The diagram in Figure 6 has removed the physical link details to focus on the modelling of the more logical resources such as compute resources, FAM resources, and logical paths and connections between them. This diagram shows only one host system, and one FAM module as Endpoints (I3 and T2, respectively) on

the CXL fabric. Endpoints are Initiators or Targets of traffic; the Switches are not shown as they only relay traffic and are not visible in the data plane.
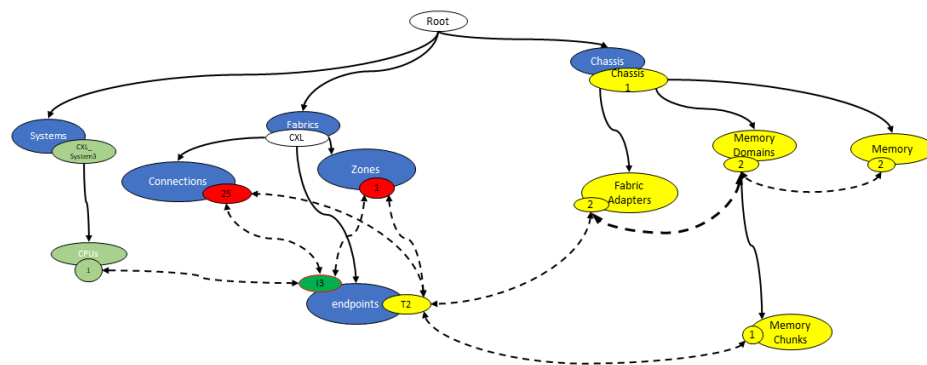
Figure 6: Basic Redfish Model of Fabric Attached Memory Device

Redfish models all fabric attached memory devices as a Fabric Adapter that moves traffic to or from the physical port attached to the fabric and transitions such traffic to accesses of memory resources 'behind' the Fabric Adapter.

The Fabric Adapter object is linked to the **Memory** sources that supply the actual memory media. The Memory sources are linked to the Redfish **Memory Domain** (memory pools) from which blocks of accessible memory capacity are allocated as **Memory Chunks**. We have already stated that for the host associated with Endpoint I3 to have a physical path through the fabric to the FAM module Endpoint T2, a Redfish Zone 1 must contain both Endpoints. However, Zones do not provide fine grained permissions for subsets of the resources within an Endpoint, as would be necessary for managing multi-tenant Endpoints. If Fabric Adapter 2 in the above figure had more than one Memory Chunk mapped to the fabric, the Redfish Zone does not contain a limitation of which Memory Chunks a given Initiator Endpoint in the Zone may access.

Thus, Redfish declares that a **Connection** specifically enables one Endpoint entity to access some or all resources presented to the fabric by another Endpoint. E.g., a specific Memory Chunk which represents only a small portion of the total memory available behind the T2 Endpoint may be named in a Connection between two Endpoints. If the host and the FAM controller are in the same Zone the fabric will route any request from the host Endpoint to the FAM Endpoint. However, if the host's request is not targeted to the Memory Chunk named in the Connection, the FAM controller is supposed to reject the request.

Discovery of physical and logical resources accessible via the fabric is necessarily a fabric-specific operation. Fabric-specific requirements for the creation and interpretation of Redfish fabric objects are listed in Appendix A (Section 7).

5. Sunfish Agents

This chapter defines the basic roles and responsibilities of a general Sunfish Agent. Detailed requirements for specific Redfish objects or properties, as well as requirements for specific fabrics are presented in Appendix A.

5.1. Common Roles, Responsibilities and Requirements

The general nature of Agent duties are explained in the overview of Agents. The following is a partial list of key responsibilities of the fabric Agent Service vs the Sunfish Core Service.

- The Agent is responsible for locating the Sunfish CORE's Event Service and issuing a form of New Fabric Event Notification to initiate Sunfish Core and Agent communications.
- The Agent is responsible for declaring the Redfish IDs of all fabric physical components because the Agent presents these objects to the Sunfish CORE. The Namespace translation between the Sunfish CORE's Redfish namespace and the Fabric Manager's (potentially) proprietary namespace is managed by the Agent.
- Logical objects created by allocating or binding resources of the physical components are given their Redfish IDs by the Service that defines them. E.g., when Clients of the Sunfish Core Service choose to POST a MemoryChunk, the Client decides what Redfish ID to use in the POST request. If the Hardware Manager had pre-configured Memory Chunks bound to specific hosts on the fabric, the Agent will assign their Redfish IDs before it makes them known to the Sunfish Core Service.
- The Agent is responsible to aggregate multiple Hardware Manager inventories into one fabric namespace since Gen-z fabrics have a flat fabric address space that must be disambiguated before the fabric can carry general data traffic. Doing this aggregation makes the Agent responsible for creating a unique Redfish ID for each unique fabric component in the aggregated fabric.
- The Agent shall provide an HTTP Redfish API to the Sunfish CORE
- The Agent shall track the status of the Sunfish Core representation of the fabric in its Agent–specific internal fabric database for all the fabric objects that it has presented to the Sunfish Core Service, and for all the logical fabric objects sent to it by the Sunfish Core Service.
  - The Sunfish Core can use GETs to the Agent's API Service to get caught up if the Sunfish Core drops
  - The Agent cannot use GETs to the Sunfish Core Service to discover what was made visible to the Sunfish Core should the Agent drop. The Agent must alert the Sunfish Core that they are potentially out of sync. The Sunfish Core must re-fetch the Agent's inventory and state and resolve any discrepancies between the new uploaded data and that data the Sunfish Core database has associated with the Agent.
  - By definition, the most recently uploaded data from the Agent, if newer (by timestamp) than what the Sunfish Core has stored is to be accepted as the latest statement of fabric state and the Sunfish Core shall amend its models to match the Agent's. See also section 4.5 for more details.

- The Agent parses Redfish requests from the Sunfish Core and
  - Evaluates the impact on the Agent's model of the fabric
  - Evaluates the impact on the Hardware Manager's state of the fabric, and creates an appropriate Hardware Manager specific request to cause the Hardware Manager to update the state of the fabric accordingly

- Returns success or failure of the Sunfish Core request along with an appropriate Redfish object that results from a successful request
- The Agent parses any Agent specific HTTP Requests. These are requests made through the Agent's client interface that target Agent internal configuration and are not meant to directly manipulate Redfish objects. Think of these as Agent administration requests.
  - Examples: Restart the Agent, sync with a Redundant Agent
- The Agent registers for events from the Hardware Manager to receive notice of dynamic changes in fabric hardware and Hardware Manager or Host software.

5.2. Security

This document will generally adhere to the security requirements and capabilities defined in the Redfish specification (Table 3), but also will include specific restrictions, policies or requirements placed upon various blocks of the Sunfish Framework or the APIs between them.

5.3. Events

This document already lists Redfish Event types and the policies to be used with them in the scenarios to register a Sunfish Agent with the Sunfish Core Service and to alert the Sunfish Core of the creation or update of a Redfish object. This section will be devoted to the process of Clients registering for events with Sunfish Core and the processes by which the Sunfish Core and Sunfish Agent cooperate to appropriately relay the requested events and any associated logs.

6. Bibliography

6.1. Overview

The following referenced documents provide important support for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

6.2. Informational references

The informational references are summarized in Table A.1.

Table A1: Informational References

Tag	Title (Version)	Author	URL
1	Redfish Message Registry (2023.3)	DMTF	https://www.dmtf.org/dsp/DSP2065

7. Appendix A: Sunfish Requirements For Redfish Modelling

7.1. Sunfish Requirements for Redfish Modeling of Fabric Attached Memory

This section defines the policies and requirements the Sunfish Architecture imposes on Clients and Agents when interpreting or creating Redfish models of Fabric Attached Memory (FAM) managed via the Sunfish Architecture.

7.1.1. Important taxonomy used within the context of discussions around FAM

Producer, Memory source, Memory resource, Memory target, Memory funder: The physical entity attached to the fabric that supplies memory resources to satisfy fabric requests for memory reads and writes. This physical entity will store or recall the data being written or read.

Consumer, Host, Compute Entity, Data Mover, Requester, Initiator: The physical entity attached to the fabric that launches a fabric request for a memory read or write transaction to the target device. This physical entity will source or sink the data contained in the fabric memory read or write transaction.

Homogeneous memory resource capacity: The first requirement for FAM capacity to be considered homogeneous is such Memory capacity shall have the same persistence or volatile property across the entire range of data locations serviced by said memory sources. A secondary consideration is not a requirement, but a strong recommendation. Homogeneous memory capacity should have a similar access latency from similar access points such that multiple threads accessing the same addresses within the capacity have consistent performance.

Fabric Attached Memory (FAM) is memory capacity that can be accessed across the fabric where:

- The source of FAM is the Producer and the target of memory requests.
- The Consumer is the initiator of a memory request and the Consumer of memory (e.g., a Host or a data move engine).

Regions vs MemoryRegions: *Regions* are sub-sets of memory media capacity in the Redfish schema for *Memory*. The CXL spec defines something completely different called a *Dynamic Capacity Memory Region* (DC Region for short) that Redfish has named a **MemoryRegion**. The *MemoryRegion* object is a subordinate of the CXL Logical Device object and therefore currently a CXL-specific object. *Regions* is a property within the *Memory* object.
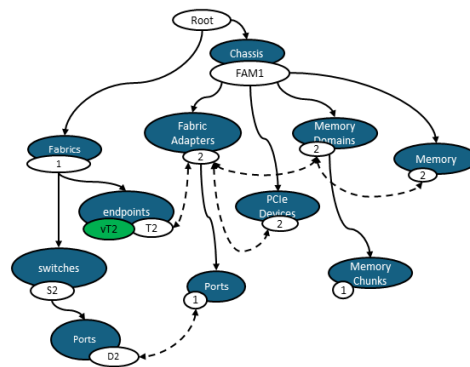
Figure 7: Basic Redfish Model of FAM Device

The Redfish bubble diagram in Figure 7 depicts the Redfish object hierarchy of a FAM object as described by the above Fabric Adapter, Memory Domains, Memory sources, and an existing Memory Chunk. See Section 4.5. for details. Note that Sunfish Core requires Sunfish Agents define a Memory Chunk to consist of homogeneous media locations from within one physical device. Larger collections of Memory Chunks from multiple physical devices are modeled as Logical Memory Regions.

Logical Memory Regions, which are DMTF Work In Progress for Redfish will be defined in a later release of this document.

7.1.2. Sunfish Requirements for Redfish Models of FAM

All fabric Agents are expected to comply with the following Sunfish interpretations and requirements when creating Redfish models of a FAM resource on any fabric. The following discussion introduces the critical Redfish objects that are intended to model FAM resources, and declares several general required policies that fabric Agents shall follow when creating these objects. FAM is memory capacity that can be accessed across a fabric.

To have a presence on a fabric, **the FAM Producer shall have at least one Endpoint on the fabric**. **A physical Endpoint shall have an associated unique fabric address that appears in the fabric transport-layer packets.** This unique fabric address does not have to be visible in the Redfish Endpoint object JSON description. However, the **Agent / Hardware Manager stack shall have the ability to translate a Redfish request targeted to a unique Redfish Endpoint object into one or more fabric-specific requests targeted at the unique fabric hardware entities.**

An Endpoint presenting physical FAM to the fabric shall have a Connected Entity link to a single Fabric Adapter. The Fabric Adapter represents the physical device that fields the fabric memory requests from the fabric and produces the response packets to return to the requester. A Fabric Adapter may have multiple ports which link into the fabric. Different fabrics may have different associations of these multiple ports to fabric IDs. A single Fabric Adapter may present more than one Endpoint on the fabric, but **any given physical Endpoint object shall not have more than one associated Fabric Adapter**. **If a physical device presents multiple physical devices to the fabric through a single fabric Port, each physical**

device shall have its own Redfish Endpoint. A Fabric Adapter may present multiple logical devices to the fabric as multiple logical Endpoints. **Each logical device shall be modelled as one and only one logical Endpoint**.

The Fabric Adapter device may or may not make known to the Hardware Manager any details about its associated chassis / enclosure, FRU, containing assembly, or other managers of such related entities. If the Hardware Manager cannot discern a specific Chassis object housing the FAM's Fabric Adapter, **the Hardware Manager/Agent stack shall create a generic FAM Chassis instance and attach the Fabric Adapter to the FAM Chassis' subordinate Fabric Adapters collection.** If the Fabric Adapter's chassis details are supplied to Sunfish by a different Agent (e.g., the chassis manager's Agent), the only clear indicator that the generic FAM Chassis instance created by the fabric Agent and the chassis described by the chassis manager's Agent are one and the same chassis is for both Agents to fill in the same Redfish Object UUID (or its equivalent) for the Fabric Adapter they each modeled. **Therefore, all fabric component instances shall posses a globally unique, vendor defined UUID or equivalent that can be read by each manager that may become involved.**

The Hardware Manager shall create the subordinate PORTS collection and instantiate an appropriate Fabric Adapter Port object for each physical PORT discovered on the Fabric Adapter. The Fabric Adapter, by definition must have at least one physical port connected to the fabric which claims this Endpoint. The Fabric Adapter may have additional PORTs that are connected to devices not currently modelled as Redfish Switches or Redfish Endpoints on this same fabric.

The Hardware Manager shall discover the entirety of the FAM resources accessible through the Fabric Adapter, and shall create appropriate homogeneous Redfish Memory resource objects to describe the available memory capacity. When the Hardware Manager discovers the FAM device, the Hardware Manager/Agent stack is responsible for breaking up all the Memory resources the adapter presents to the fabric into homogeneous memory resource pools. Some FAM devices may declare their memory resources as specific commercially available memory devices (e.g., DDR5 DIMM devices, Optane NVDIMMs). Some FAM devices may declare their memory resources as different quantities of generic memory media types (e.g., volatile RAM, persistent RAM, persistent block-mode storage). Homogeneous Memory resources may be modelled using any applicable Redfish Memory object. Non-homogeneous Memory resources shall be modelled as an appropriate number of homogeneous Memory resources.

After creating the appropriate homogeneous Memory resource objects for a Fabric Adapter, **the Hardware Manager/Agent stack shall create one or more homogeneous Memory Domains linked to appropriate collections of the Fabric Adapter's Memory resource objects.**

Any Fabric Adapter which acts as a producer of FAM resources to the fabric thus shall have navigation links to one or more *homogeneous* Memory Domain objects.

If the Hardware Manager/Agent stack has an a priori plan that dictates how one or more of a FAM module Fabric Adapter's Memory Domains are to be split up into individual Memory Chunks, **the Hardware Manager/Agent stack shall create the desired pre-defined Memory Chunks**. If there are no a priori plans that dictate allocation of Memory Chunks from the various Memory Domains, **the Hardware Manager/Agent stack shall not create the Memory Chunks collection or any default Memory Chunk instances.** Their existence at the first appearance (creation time) of a FAM module will imply such memory allocations are reserved and not available to Clients of the Sunfish Redfish Service.

7.1.2.1. Sunfish Interpretation of Redfish FAM Object Properties

The Redfish properties for all the objects which Sunfish recognizes as describing FAM sources or producers on a fabric are defined in the Redfish Resource and Schema Guide (Table 3). However, sometimes the flexibility of the Redfish definitions can lead to confusion or inconsistencies in their use by different creators. Therefore the Sunfish Framework is dictating specific rules for meaning *and usage* of some properties. Some such rules apply to properties associated with the general object outside the context of a specific fabric (e.g., any of Ethernet, Infiniband, SAS, etc) and some rules apply to the general property within the context of a specific fabric (e.g., only CXL, only Infiniband), and some rules apply only to properties or whole objects that are unique to a specific fabric.

In the following tables, the column header definitions apply:

- **Property:** Nominally just the name of a Property as found in the Redfish Resource and Schema Guide tables under the 'Property' column. In the Redfish guide, rather than a simple name assigned to a 'string', 'boolean', or 'ordinal', some properties are really the JSON descriptions of arrays of URIs or key, value pairs, etc. To save space, our 'Property' column will include a 'file system directory format' for such nested objects.

- For example, the structure:

```
Links {
  Endpoints [
    { @odata.id:  },
  ]
}
```

would be depicted in the following Sunfish tables as:

Links/Endpoints

- **Opt/Req:** Redfish makes many properties of objects optional. Sunfish is more particular about the presence of some properties. The value in this column indicates the requirement of this Property in the Sunfish Framework:
  - **Optional:** (Opt) Sunfish does not require this Property be present, but when it is present any requirements in the Notes column apply.
  - **Required:** (Req) Sunfish requires this Property be present for the object to meet Sunfish's definition of 'complete'. If this Property is not present, the Sunfish or the Sunfish Agents may reject any POST, PATCH or PUT requests directed to this object because a complete description is required to perform the request properly.
  - **Conditionally Required:** In some situations this Property is Required. Often applied to 'Links' to related objects. E.g., CXL SLD or MLD devices need one or more CXLLogicalDevices linked to the FabricAdapter which represents the physical Endpoint to the fabric.
  - **Prohibited:** In some situations this Property is Prohibited. The value may be meaningless or ambiguous. In the case of links, some navigation links would imply improper relationships or circular definitions.
- **Client/Agent:** Indicates the entity that is normally expected to declare the value of the given Property. This column is only a guideline. Normative requirements, if any, may appear only the Notes column.

- ○ **Client:** indicates that the Sunfish Client either establishes the original value of the Property, or is allowed to over write existing defaults or values originally established by the Agent / Hardware Manager stack.
  - ○ **Agent:** indicates that the Agent / Hardware Manager stack is expected to supply the original value of this property when it was created and passed into Sunfish by the Agent.
  - ○ **Agent crawlout:** indicates that the Agent / Hardware Manager stack is required to supply the original value of this property when the Agent passes the original object to Sunfish. This Property is usually a hardware declared value or is a value established by the Hardware Manager during the fabric crawl out and base configuration phase.
- **Notes:** This column describes the Sunfish specific extension, explanation, or interpretation of the purpose of this Redfish property. If the definition is the same as found in the Redfish Resource and Schema Guide, then this column should be indicating addition details about how Sunfish will use this value, or describe policies related to processing specific values or changes of values. *This column may contain normative requirements*.

In the following tables the term "**UUID-formatted**" refers to a 16-byte identifier formatted as a UUID string. The 16-byte value does not need to meet the IEEE definition of a UUID (IE, it is not a Sunfish requirement that this identifier meet the IEEE UUID requirements). To meet the Sunfish requirements, the 16-byte value shall be unique within the scope of the Redfish URI namespace listed in the Notes column.

Fabric Adapters: When creating a Redfish FabricAdapter to describe FAM, the Hardware Manager / Agent stack shall use the following interpretations for specific Properties.

Property	Opt/Req	Client / Agent	Notes
FabricType	Req	Agent crawlout	The *configured* fabric type of this fabric adapter. Some NICs support multiple protocols. **This property shall declare the FabricType currently configured. If the adapter is re-configured to support a different FabricType, this Redfish instance shall be deleted and a new one created with a different Redfish URI**.
Links to MemoryDomains	Req	Agent crawlout	Links to MemoryDomain objects which may be used as memory pools sourced by this FabricAdapter. For CXL FAM modeling, each MemoryDomain is assumed to be made up of *homogenous* Memory sources.

Property	Opt/Req	Client / Agent	Notes
UUID	Req	Agent crawlout	A UUID string that uniquely identifies this FabricAdapter instance within the scope of the Fabric or Fabrics on which this adapter represents an Endpoint. Redfish schema lists this property as a UUID. Sunfish Framework uses this value to correlate Fabric Adapter hardware instances and resolve duplicate models reported by different Agents. For Sunfish's purposes, this value only needs to be UUID-formatted. Some Agent / Hardware Manager stacks will place a IEEE compliant UUID string here. Some will use a 16-byte combination of Vendor ID, Serial Number, and other unique hardware values.

MemoryDomains: When creating a Redfish Memory Domain to describe FAM, the Hardware Manager / Agent stack shall use the following interpretations for specific Properties.

Property	Opt/Req	Client / Agent	Notes
InterleavableMemorySets/MemorySet	Req	Agent crawlout	Links to all Memory resource objects that fund this MemoryDomain. **All such Memory resources shall be homogeneous for CXL FAM.** The CXL Agent/Hardware Manager stack is required to send only homogeneous MemoryDomains to Sunfish Core. **Non-homogeneous physical Memory sources shall be modeled as multiple homogeneous Memory sources.**
Links/CXLLogicalDevices	Cond Req	Agent crawlout	Array of links to all CXLLogicalDevices that may present portions of this MemoryDomain as resources of the Logical Device. This is a CXL property specific to the CXL SLD and MLD devices. **The Agent/Hardware Manager stack shall create a CXLLogicalDevice for every logical device supported by the device.**

Property	Opt/Req	Client / Agent	Notes
MemorySizeMiB	Cond Req	Agent crawlout	Redfish does not require MemoryDomains to be homogeneous. For Sunfish, this is the **total size of the homogeneous MemoryDomain in mega-bytes. If present, MemorySizeMiB shall represent available capacity of all sources in the *homogeneous* InterleavableMemorySets. This field shall not be present if the InterleavableMemorySets are *heterogeneous.*** Sunfish clients will not consider this MemoryDomain when taking inventory of FAM capacity.

MemoryChunks: When creating a Redfish Memory Chunk to describe allocated FAM, the Hardware Manager / Agent stack shall use the following interpretations for specific Properties.

Property	Opt/Req	**Client / Agent	Notes
AddressRangeType	Req	FM crawlout	Basic memory type (Block, PMEM, Volatile) of entire chunk. **This property shall be consistent with the properties of the Memory sources making up the MemoryDomain from which this chunk is allocated.**
AddressRangeOffsetMiB	Opt	Client / Agent	Offset of the MemoryChunk in MiB from base (0) of the MemoryDomain as set by the creator of the chunk . This offset may or may not be meaningful to the other clients or to the actual device. This property is optional for CXL FAM.
DisplayName	Cond Req	Client / Agent create	Redfish defines this as "A user-configurable string to name the memory chunk." When a Memory Chunk is created by the Client, there are no restrictions on the choice of DisplayName imposed by Sunfish Framework. However, **if the Agent / Hardware Manager creates the Redfish MemoryChunk, the Sunfish requirement is that this property shall be assigned a 16-character (16-byte) UUID string in the Redfish UUID format.**

Property	Opt/Req	**Client / Agent	Notes
MediaLocation	Req	Client create	For MemoryChunk instances of FAM, **this value shall always be set to "Local"**. (Options are Local, Mixed, Remote) When modeling FAM MemoryChunks for Sunfish, **such chunks shall be allocated from Memory sources to be co-located within a single MemoryDomain,** and thus 'local' to the associated FabricAdapter(s). Sunfish uses the Redfish Logical Memory Region (currently a Redfish WIP) to model aggregation of multiple MemoryChunks (possibly from multiple Fabric Adapters) into larger FAM pools.
Links/CXLLogicalDevices	Cond Req	Client create	Array of links to any CXLLogicalDevices that use this MemoryChunk to fund Logical Device capacity. **This property shall be present in Sunfish MemoryChunk objects on CXL fabrics.**
InterleaveSets/ Memory/ OffsetMiB RegionID SizeMiB	Opt	Agent create	Optional For MemoryChunk instances of FAM, Since FAM MemoryChunks are created from Memory resources 'local' to the FabricAdapter the Client may not have the ability to specify the exact Memory resources that will fund a MemoryChunk. The Agent / Hardware Manager stack may return these details if it can extract them from the hardware. **A Sunfish CXL FAM MemoryChunk shall only be allocated from a single RegionID of a multi-region Memory source.** This prevents need for multiple passphrases associated with a single MemoryChunk.

Memory: When creating a Redfish Memory source (Redfish uses the term Memory device, but that is misleading) to describe FAM media, the Hardware Manager / Agent stack shall use the following interpretations for specific Properties. There are many other Properties of a Memory source defined in the Redfish schema. So far, the following have Sunfish specific distinctions. There will be more as Sunfish and Redfish both evolve.

Property	Opt/Req	Client / Agent	Notes
BaseModuleType	Opt	Agent crawlout	This property is optional for FAM components. If the FAM module uses a specific type of memory device to supply the media capacity, and the type is known to the Hardware Manager / Agent stack, this property may be filled in.

Property	Opt/Req	Client / Agent	Notes
CapacityMiB	Cond Req	Agent crawlout	Memory capacity in MiB. **This Property shall be present if and only if all capacity in the resource object is _homogeneous_.**
ErrorCorrection	Opt	Agent crawlout	If the error correction method supported by the memory source for a FAM module is known, the Agent / Hardware Manager stack shall include this property.
OperatingMemoryModes	Opt	Agent crawlout	Array of strings describing the memory modes supported by this Memory resource. (Block, PMEM, Volatile) If more than one mode is listed in the array, this is a multi-mode Memory resource. The mode the resource is currently configured for is dictated by the Regions/ MemoryClassification property.
Links/ MemoryMediaSources	Prohibit	Agent crawlout	Redfish defines these as links to MemoryChunk resources that provide media to this Memory resource. These links allow Memory sources to be linked to 'remote' MemoryChunks. FAM Memory media sources are all 'local'. Thus, **models of FAM Memory resources shall not have these MemoryMediaSources arrays in their Links object**.
Links/ MemoryRegionMediaSources	Prohibit	Agent crawlout	Redfish defines these as links to (CXL Dynamic Capacity) MemoryRegion resources that provide media to this Memory resource. This allows Memory sources to be linked to 'remote' MemoryRegions. Since FAM Memory resources are all local, **models of FAM Memory resources shall not have these MemoryRegionMediaSources arrays in the Links object.**
Regions	Req	Agent crawlout	Array of Region descriptions that define homogeneous sub-ranges of memory capacity within this Memory resource. These are Regions in the Redfish schema for Memory. The CXL spec defines something completely different called a Dynamic Capacity Memory Region (DC Region for short) that Redfish has named a MemoryRegion.

Property	Opt/Req	Client / Agent	Notes
Regions/ MemoryClassification	Req	Agent crawlout	Basic memory type (Block, PMEM, Volatile) of entire region. This property dictates the corresponding AddressRangeType property of the MemoryDomain funded by this Region. **A homogeneous Sunfish FAM Memory resource shall have the same MemoryClassification in all Regions.**
Regions/RegionID	Req	Agent crawlout	Unique region ID associated with this Region. Unique value across all associated Memory resources used to model the memory capacity of a non-homogeneous memory source associated with a FAM FabricAdapter.
Regions/SizeMiB	Req	Agent crawlout	Size of this Region in MiB. Sunfish clarification: This value represents physical capacity present, not potential capacity not yet installed.
Regions/OffsetMiB	Opt	Agent crawlout	Offset of the Region in MiB from base (0) of the Memory resource. This value is device specific. Some fabrics do not make such an offset visible outside the memory device. This offset, if present, may or may not be meaningful to the Sunfish client, the Agent / Hardware Manager or the actual device. It may be used in an Agent-specific manner for bookkeeping.

Endpoints: When creating a Redfish Endpoint to describe FAM, the Hardware Manager / Agent stack shall use the following interpretations for specific Properties.

Property	Opt/Req	Client / Agent	Notes
ConnectedEntities / EntityLink	Req	Agent crawlout	Links to Redfish modeled entities connected to the fabric at this Endpoint. **For a FAM physical resource (the hardware with the port on the fabrics, e.g. a CXL MLD) the ConnectedEntity shall link to a FabricAdapter object.**
ConnectedEntities / EntityRole	Req	Agent crawlout	**For a CXL FAM device, the EntityRole shall be "Target" or "Both". For a CXL host, the EntityRole shall be "Initiator" or "Both".**

Property	Opt/Req	Client / Agent	Notes
ConnectedEntities / EntityType	Req	Agent crawlout	For a CXL FAM device, the Endpoint ConnectedEntity is either a CXL Logical Device (a virtual endpoint) or a 'FabricBridge' (a Fabric Adapter, the physical object). **Sunfish Agents shall map FAM devices to Endpoints with ConnectedEntity == FabricBridge**
HostReservationMemoryBytes	Opt	Agent	The amount of memory address space that should be reserved by any initiator planning to target memory mapped resources on this endpoint. **If present, this Property shall be set to 0 if this Endpoint has no FAM capacity to map to this fabric.**
Links/ConnectedPorts	Cond Req	Agent	Sunfish expects each Endpoint is associated (directly or indirectly) with a single physical port of the physical device that presents resources to the fabric. This single physical port may be linked physically to a single port of another fabric component. **If the physical link exists this array shall contain the Redfish URI of the port of this other component.**
Links/LocalPorts	Req	Agent	**The LocalPorts array shall contain the Redfish URI of the *physical port* of the ConnectedEntity object.** For a FAM module mapped to the fabric, this will be the port of the FabricAdapter that sources the FAM onto this fabric. If the ConnectedEntity is a CXL Logical Device or some other logical construct behind the FabricAdapter, the LocalPorts URI is still the Redfish URI of the port of FabricAdapter.

Ports: When creating a Redfish Port to describe fabric components, the Hardware Manager / Agent stack shall use the following interpretations for specific Properties.

Property	Opt/Req	Client / Agent	Notes

Property	Opt/Req	Client / Agent	Notes
CXL/ CurrentPortConfigurationState	Cond Req	Agent crawlout	CXL switch and device ports have several CXL-specific properties. The CurrentPortConfigurationState informs clients whether this CXL Port is in a transitional state, a disabled state, or a configured state such as a CXL Upstream Port (USP), a CXL Downstream Port (DSP), or a CXL inter-switch link port (FabricLink). Transitional state values such as BindInProgress and UnbindInProgress are required if the Agent creates a task to modify the hardware port's configuration state.
CXL / MaxLogicalDeviceCount	Cond Req	Agent crawlout	**If the Port is physically part of a CXL device that supports CXL Logical Devices this property shall indicate the maximum number of CXL Logical Devices that can be addressed through this Port. The Agent / Hardware Manager shall produce this number of CxlLogicalDevice instance models.**
Links / ConnectedPorts	Cond Req	Agent crawlout	**If this Port is attached to another Port, the Agent / Hardware Manager shall include this link to the Agent / Hardware Manager's Redfish representation for this partner component's Port. If this Port's manager is not the manager of the link's partner component, this manager shall create a placeholder model of the link's partner component and Port.**
Links / ConnectedSwitches	Cond Req	Agent crawlout	**If this is a Device or Switch Port connected to a / another Switch, the Agent / Hardware Manager shall include this link to the Agent / Hardware Manager's Redfish representation for this partner Switch. If this Port's manager is not the manager of the link's partner Switch, this manager shall create a placeholder model of the link's partner Switch.**

Property	Opt/Req	Client / Agent	Notes
ID	Req	Agent crawlout	All Redfish Port objects require a string value that is the Redfish ID of the Port. This Redfish Port ID string is not required to contain or imply the physical port number in the hardware. Redfish Clients use the PortID Property to discover the physical port number.
PortID	Cond Req	Agent crawlout	This is the port number by which this hardware component internally addresses this port. **If this is a CXL Port description, this Property shall be included.**
RemotePortID	Cond Req	Agent crawlout	During CXL *fabric link* training, CXL fabric Ports exchange an 8-bit value denoting their port number. This is the port number received from the link partner component during link training. If this Port object is not a CXL fabric port, the Agent or Hardware Manager must discover the RemotePortID via out of band means. The expectation is all RemotePortIDs are known for all components within a Hardware Manager's domain. Further, the expectation is CXL fabric links (switch to switch links) are used between Hardware Managers' domains. **If this is a CXL fabric Port description, this Property shall be included if the link has been trained. This Property shall not be included in Port descriptions submitted to Sunfish Core if this value is not known by the Agent / Hardware Manager crawlout process.**

Zones: When processing a Redfish Zone to describe fabric paths that enable traffic between Endpoints, the Hardware Manager / Agent stack shall use the following interpretations for specific Properties.

Property	Opt/Req	Client / Agent	Notes

Property	Opt/Req	Client / Agent	Notes
DefaultRoutingEnabled	Req	Client	The meaning of this property within the Sunfish Framework is still unclear. The definition of this property may vary by fabric type. The following restriction is subject to change: **In a CXL fabric, Sunfish requires that the Default Zone shall set this Property to 'False' so that Endpoints in the CXL Default Zone are isolated from one another until a Client moves them to another Zone. **
Links/InvolvedSwitches	Cond Req	Client	If there are switches in the paths between producers and consumers of fabric resources, **the Client shall include all switches in the paths between Endpoints that are allowed to Zone traffic in this Zone.** Should a switch not be a member of this array, the Agent / Hardware Manager **shall not** configure it to flow traffic for this Zone. Should this Property not be present, the Agent / Hardware Manager is allowed to configure any number of paths using any relevant switches to flow traffic amongst the Endpoints of this Zone.

Connections: When processing a Redfish Zone to describe fabric paths that enable traffic between Endpoints, the Hardware Manager / Agent stack shall use the following interpretations for specific Properties.

Property	Opt/Req	Client / Agent	Notes

Property	Opt/Req	Client / Agent	Notes
DefaultRoutingEnabled	Req	Client	For ZoneType == 'Zone of Endpoints' or 'Zones of Zones', setting this flag to 'True' informs the Agent / Hardware Manager stack that the Client is not going to define the routing of paths between the Endpoint members of this Zone, and that the Agent / Hardware Manager stack is to use a fabric-specific default routing algorithm to do so. For these ZoneTypes, if this Property is set to 'False', the Agent / Hardware Manager stack and the Client are responsible to coordinate a custom routing policy. For ZoneType == 'Default Zone', Sunfish assumes that if DefaultRoutingEnable == 'True', Agents / Hardware Managers are expected to route traffic to/from any / all Endpoints in the Default Zone. **In a CXL fabric, Sunfish requires that the Default Zone shall set this Property to 'False' so that Endpoints in the CXL Default Zone are isolated from one another until a Client moves them to another Zone. **
Links/InvolvedSwitches	Cond Req	Client	If there are switches in the paths between producers and consumers of fabric resources, **the Client shall include all switches in the paths between Endpoints in a Zone that are allowed to pass Zone traffic. Should a switch not be a member of this array, the Agent / Hardware Manager shall not configure it to flow traffic for this Zone.** Should this Property not be present, the Agent / Hardware Manager is allowed to configure any number of paths using any relevant switches to flow traffic amongst the Endpoints of this Zone.