DMTF®
distributed management task force, inc.

Redfish Device Enablement Proposals Work-In-Progress

www.dmtf.org

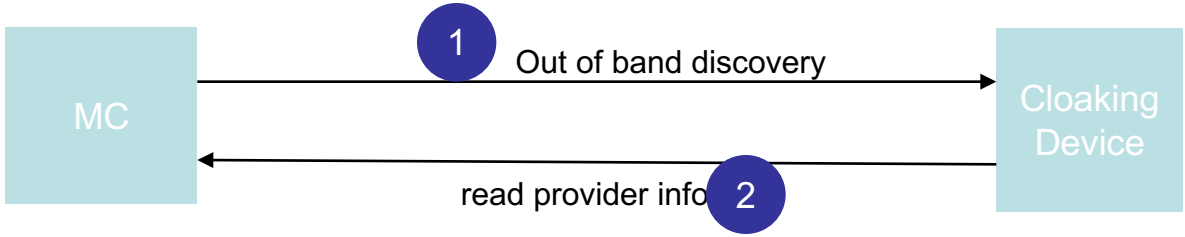
Disclaimer

- The information in this presentation represents a snapshot of work in progress within the DMTF.
- This information is subject to change without notice. The standard specifications remain the normative reference for all information.
- For additional information, see the Distributed Management Task Force (DMTF) website.

PLDM

- By calling this Redfish for PLDM, this would be a new message type for PLDM
  - This would be 06
- The benefit of this is you pick up MCTP and all the other transports and mappings already done in the PMCI WG

Discovery of Redfish Providers



MC needs “Provider info” for each resource found during discovery.

- MC Requests Provider Info
- MC validates response based on its capabilities.

Provider responds with standardized message containing information about the provider. Things like:

- Versioning of the data model language.
- Schema Types and versions supported
- Instances supported with identifiers for those instances
- Information about the operational semantics supported by the device.

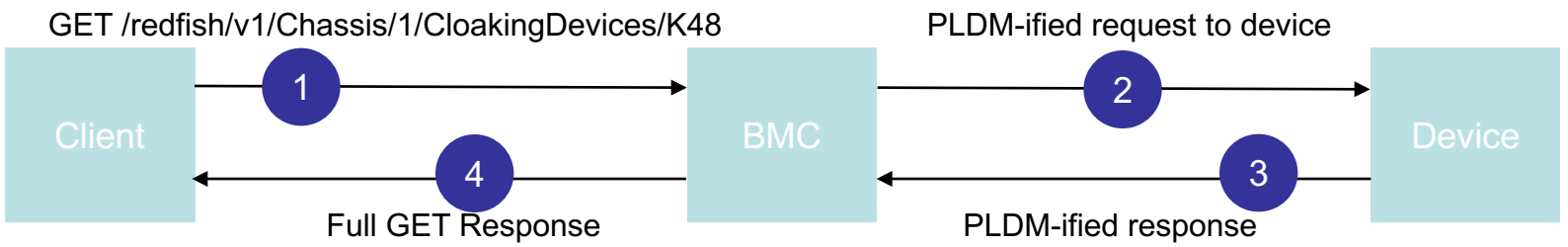
PLDM base has

- discovery of versions of the data modeling language
- commands of a device
- Would need additional information (message type specific) to cover the schema type and version

Provider handshake Info

- Version for this interface
  - Version of the interface. This should be in a fixed location for every version of this standard going forward.
- Payload SAR (Segmentation and Reassembly)
  - Max packet size and number for this protocol. Redfish packets, even compressed can be pretty big. SAR will need to be part of the architecture. Devices can always answer with a small packet size.
    - PLDM FW update did something like this. There is also the medium specific built into MCTP. So leveraging the PLDM FW update spec may help.
- Resource Map
  - Resource Count
  - Individual Resource Identifier for each resource.
  - Class & Version for each resource
  - Number of Outstanding Operations & Async Support indicators
  - OPs supported (CRUFD)
  - OPs payloads supported (JSON/BEJ)
  - Privileges (if any)
  - Eventing Mechanism Info
  - Navigation/metadata property info capability (see upcoming slides)
  - Power level specific capabilities
- CRC32
  - We would certainly need this for the OPs packets, probably ought to have it here.
  - Would be used for SAR across all packets. Need one for headers?
    - PLDM SMBIOS Table Transfer method has something we can lift for this.

What about Redfish Operations?




```
GET /redfish/v1/Chassis/1/CloakingDevices/K48

{
    "@odata.type": "#CloakingDevice.v1_0_0.CloakingDevice",
    "Cloaking": true,
    "UEFIDevicePath": "/PciRoot(0x0,0x0)/PciDevice(0x0,0x1)",
    "Diags": {
        "@odata.id": "/redfish/v1/Chassis/1/CloakingDevices/K48/Diags"
    }
}
```

We have some choices to make here:

Read "<ResourceId>" from device

Data Properties filled in by device

- could straight JSON over PLDM or PLDM BEJ (tbd)

MetaData Properties

- Could be filled in by device (if it knows them, which it could be told on registration)
- Could be substitution variables filled in by MC.

So let's lay these out before deciding

What's in Redfish Requests?

How can Redfish present a consistent data model for out-of-band devices?

- Mapping Redfish requests to MCTP means first taking a look at the different kind of properties in a Redfish payload.
- Redfish Resource Data consists of:
  - Data Properties (e.g. MACAddress) – actual data the client is interested in
  - Metadata Properties (e.g. @odata.type) –enables self description and resource management
  - Navigational Properties (e.g. @data.id) – enables dynamic data model discovery

- Assume a device knows how to handle r/w of Data Properties
- How would a device know its UEFIDevicePath and other contextually sensitive Metadata Properties?
- How would a device know the URI values to place in relative links?

```
GET /redfish/v1/Chassis/1/CloakingDevices/K48

{
    "@odata.type": "#CloakingDevice.v1_0_0.CloakingDevice",
    "Cloaking": true,
    "UEFIDevicePath": "/PciRoot(0x0,0x0)/PciDevice(0x0,0x1)",
    "Diags": {
        "@odata.id": "/redfish/v1/Chassis/1/CloakingDevices/K48/Diags"
    }
}
```

- Must accommodate:
  - Operations
    - GET
    - POST (create to collection)
    - DELETE (from collection)
    - PUT/PATCH
    - POST (custom action)
  - Other:
    - Need a way of simulating ETags

How do we encode the payload?

The decision about the payload of the data will drive some of the decisions about the header, trailer & packet-layout.

Allow JSON Text:

- Easy on the MC
- Difficult for some (most) providers.
- Some variable substitution may still be needed.
- Arguably impossible over I2C

Allow Binary Encoded JSON:

- Slightly more work on the MC
- Much easier on Providers.
- Variable substitution can be done during the Binary to JSON conversion in the MC (if any).

Allow both and let provider negotiation/payload header

- Initial request for MC could indicate what it allows.
- Response from provider could indicate what it supports.
- Could use bits in the header to indicate what is in this packet.
- Results in a more complicated set of possibilities in the ecosystem (is it worth the trouble?)

Need to pick one, the other or both.

- **Proposal is to put some bits in the payload that include encoding type and only have Redfish BEJ/SFLV encoding in the first draft of the spec.**

How do we encode the metadata?

The decision about the metadata representation will drive some of the decisions about the header, trailer & packet-layout.

MC does the substitution:

- Standardize on specific strings or Binary value types for metadata.
- Easy for providers (fixed)
- MC could substitute during the binary to JSON conversion (little cost) for binary payloads
- Would force MC to “crack packets” for JSON payloads (extra cost)

Provider does the substitution:

- Provider given values for substitution at Registration/discovery negotiation.
- Much harder on Providers. Not possible for some (most?)
- Easy on the BMC for Text, allowing for pass through.
- No “extra work” for BMC for Binary encoding.

Allow both and let provider negotiation/payload header

- Initial request for MC could indicate what it allows.
- Response from provider could indicate what it supports.
- Could use bits in the header to indicate what is in this packet.
- Results in a more complicated set of possibilities in the ecosystem (is it worth the trouble?)

Need to pick one, the other or both.

- **The initial version of the specification clearly has to cover the “MC does the substitution” case.**

How do we get the schema?

This is a fundamental decision point too. The MC has to have a copy of the Redfish schema for the device in order to expand the payloads.

- The Device/Provider has the schema
  - It could be encoded in a standard method to make it small
  - It is a way to guarantee that the MC would be able to support the device
  - Devices with multiple “personalities” would need to carry all possible schemas.

- The MC gets the schema some other way (embedded, internet,)
  - We cannot guarantee that the MC will have access to the
  - The MC could carry them all, but this would require a “firmware update” which is beyond the requirements.

- It could be both, but probably not either.

The only way that seems to meet the requirements is for the device firmware to reference the information

  - The schema could be loaded with FW, Driver, package or other mechanism.
  - Maybe a way to load the schema into the MC (similar to installing FW).
  - We need a compact/encoding way to expand the binary version to something the MC can use for expanding the Redfish BEJ/SFLV

Metadata Substitution

If the MC does the substitution, we need an encoding method so that it knows what to encode where

For the JSON (text) payload case, the following could be used for the MC substitution.

- Surrounded by “%” symbols
- Can be mixed with static text (e.g. “%UEFIDEVICEPATH%/Stuff”)

- %TYPE% - the type/version of the resource (for @odata.type)
- %LINK.<resource-name>% - the BMC-assigned URI of a provider defined resource
- %LINK.SELF% - the BMC-assigned link to the current resource
- %LINK.SYSTEM% - the BMC-assigned link to the computer system resource of context
- %LINK.CHASSIS% - the BMC-assigned link to the chassis resource of context
- %UEFIDEVICEPATH% - the UEFI Device Path of the instance of the device known to BIOS/BMC but not to device
- %INSTID% - the instance ID portion of the provider ID

For the Binary encoding, we would use type fields to indicate it is a metadata variable and the value to indicate which one.

For provider registration where the provider does the substitution, the same method/encoding can be used at registration time (should we be interested in pursuing that).

Management of Devices via Binary Encoded JSON (BEJ)

Design Philosophy

- Support full expressive range of JSON
- Want something with the positive attributes of something like ASN.1 but applicable to Redfish Schema definitions for encoding.
  - Compact, very little overhead from non-data elements
  - Names of data items are not encoded; assumption is that recipient is familiar with structure and can align things as needed
    - Use the Schema in a consistent format for encoding/decoding at the MC.
    - Provider writer can just use the element Ids for the provider code (no need for dynamic encoding/decoding at the provider).
  - No need for multiple encoding techniques
    - Same algorithm for all devices (NICs, Storage, Cloaking Device…)

Encoding of Nonnegative Integers

- Use same methodology as ASN.1 basic encoding rules (BER)
  - Low seven bits of a single byte handles values of up to 128 (high bit zero in this case).
  - High bit set means that the remaining seven bits specify the number of bytes needed to encode the integer
  - Integers encoded big endian (a byte of higher significance is sequenced before a byte of lower significance)
  - Supports values between 0 and 2^{56}-1
- This encoding is used for **all Nonnegative Integers** in the encoding.
  - Sequence numbers, etc.
  - Need a decision on endian-ness. PLDM is inherently little endian, so that makes a lot of sense.

Basic Unit of Encoding: SFLV

- S is Sequence Number
  - 0-based index of a field in its enclosing set
  - Fields are <stuff> in JSON { <stuff> } as defined by schema.
    - More on this in “Names of Fields” in 2 slides…
  - Sequence number is 0 for primitive types
  - Encoded as “Nonnegative Integer”; usually one byte
  - Maybe use the MSB for Annotations here (see slide 25 for more details)
- F is format
  - 6 bits (0..5) for basic type give 64 possibilities
    - By my count, we only really need 9
    - Boolean, integer, string, set, binary, timestamp, null, float, bitstring, array, choice
    - Rest reserved for future use and to fill in things overlooked
    - Maybe use a bit here (5) to indicate an Annotation. Does limit types to 32.
  - Bits 6,7 for delayed-binding substitutions
    - 00: normal data
    - 01: ignore the type for the rest of this byte; non-negative integer encoding specifies entry in substitution lookup table
    - 10: substitution lookup table entry with textual description for substitution
    - 11: substitution lookup table entry with substitute binding supplied (encoded per bits 0..5)

Basic Unit of Encoding: SFLV (cont.)

- L is Length
  - Encoded as a “Nonnegative Integer”
  - Size in bytes of the data for this unit
    - Everything after “SFL”
  - Generally follow ASN.1 BER, no reason to reinvent the wheel here
  - Issue: devices could/would need the lengths for all values to prevent buffer overrun and proper allocation
    - Possible solution: include in the encoding of the schema on the device (which it give the MC) the max length for each property. This could apply to max array size too.
- V is Value
  - Encoded in fashion specific to the format
  - Generally follow ASN.1 BER, no reason to reinvent the wheel here
  - Counts for sets, arrays encoded as a nonnegative integer prefix to the raw data contents

Names of Fields

- Full JSON gives name values for fields as part of the textual encoding
- BEJ substitutes them for a single binary value that maps to the schema
  - Unlike every other binary encoding of JSON out there.
- How to restore them going from JSON->BEJ->JSON?
  - Need ability to uniquely map BEJ encoded data to the Redfish schema it's associated with
  - Structure GUID, or some equivalent, encoded as part of the structure.
  - BEJ decoder can then walk the schema in parallel with encoded data so it knows what it's looking at
    - Can supply field names from JSON schema as we go
    - Proof of concept does this in about 20 lines of code
  - Propose Open Source code to do the encoding/decoding to ensure we have interoperability.

Pointers

- Idea: tree prefix to a point of interest
  - Can be to a set, an array, a leaf node
- Encoded as a list of sequence numbers for the tree nodes encountered in the traversal to the point of interest
  - Encode as non-negative integers
  - Typically single bytes
  - Generally, very shallow trees; expect to be able to encode pointers in 4-8 bytes

URIs

This is needed because a local device might know about other local resources where the Redfish payload would point

- Examples: Volumes/Disks, ACD endpoints

- MC translates pointers to URIs based on its resource table as part of the BEJ expansion.
  - Because Redfish is a hypermedia transport, the MC can locate the resource URI at the root URI appended with the resource ID.

Headers and Substitution Lists

If we are going with PLDM, this may not be necessary since you could use multiple commands but we *could* do this in one command…

- (everyone shakes head “no”).

Header is an encoding of JSON collection of fields:

```
header : {
 RedfishVersion : “1.0”,
 ManagementTarget : { provider: <some sort of descriptor here> },
 Command : {
  resource: <descriptor here>,
  resourceSchemaVersion: “1.0”,
  actionVerb: <“read”, “write”, “insert”, “delete”, “create”, …>,
  operationID: <operation-unique identifier for tracking purposes>
  statusCode: 1,
 }
}
```

Substitution List

- Substitution List is an encoding of JSON array of substitutions

```
Substitution list : [
 0: “description”,
 1: “name”,
 2: <replacement value>,
 3: “id”,
 4: <replacement value>,
 …
]
```

- Encoding, bits 6,7 makes clear which are descriptions and which are replacement values: 10 for description, 11 for replacement

Verbs and Command Formats

With PLDM and single format, this may not be 100% relevant now since we will need individual PLDM Message commands for CRUFD but:

- General format for a command is:

  [Header][Substitution List][Pointer][Payload]

- All but pointer are BEJ encodings of JSON, so have length defined within. This makes it easy to tell where one field ends and the next begins.
- Pointers are length-prefixed to define where the payload starts.

Verb	Header	Substitution List	Pointer	Payload
Read (GET)	√	√	√	Null
Write (PUT/PATCH)	√	√	√	Data updates
Insert (POST)	√	√	√	Data to insert
Delete	√	√	√	Null

Special Considerations: OEM & Annotations

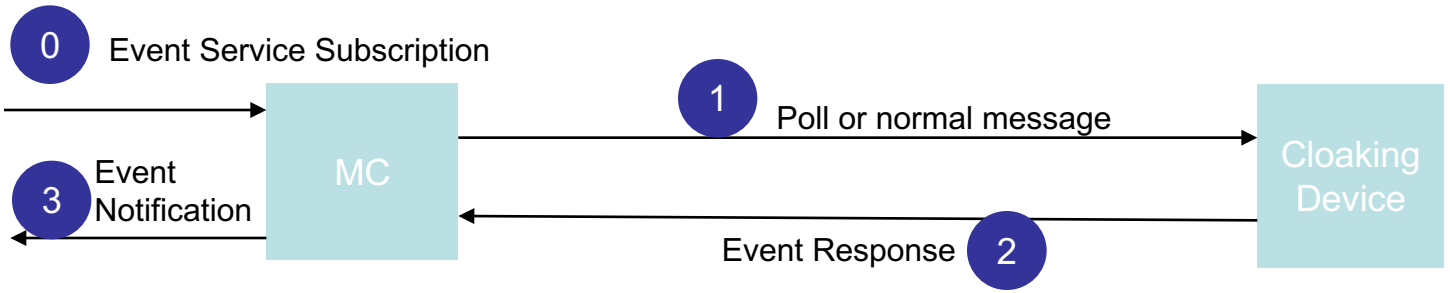
- OEM sections and Annotations “add” to the schema, making a guaranteed, consistent encode/decode problematic.
- Propose treating them differently
  - OEM: treat them as additional resources and they are included as part of the discovery as separate resources.
    - That means the MC will have to have/cache the OEM section in order to recreate the JSON payload for the resource containing OEM.
    - You can either expand them in the resource or put a link to the separate resource in the oem section.
  - Annotations: We need a way to solve the fact that annotations can be included any time.
    - Only DMTF & OData Annotations are allowed.
      - Other Annotations and annotations in other locations are not allowed unless specifically included in the schema definition
    - Always assume standard odata as well as Redfish annotations are available at the root and make them part of the encoding for every schema.
      - This blows the 256 byte schema definitions pretty quickly.
    - As an alternative, we could use the MSB of the sequence number or a bit in Format to indicate that this is an annotations and use a canonical list of the available standard annotations.

Errors, Eventing and Tasks

Error mapping

- Need to extend the Redfish message registry to cover the error scenarios that may occur here
  - We need to map the MCTP, PLDM & Redfish Device Enablement errors to Redfish Message Registry.
  - These will need to be part of the standard DMTF registry and/or HTTP (some are in there already and should be reused).
- Also need a standard alerting registry
  - We need to develop the alerts as well since that will be a message we define. So we need the list of alerts.
    - Payload could be an encoding of a Redfish Event, SFLV style.
  - DSP8007 exists for CIM Alerts. Is that the right place to start for creating a new event message registry? This registry needs to be coordinated with SPMF.

Eventing for Redfish Providers over PLDM



MC sends
1) Periodic Poll
2) Normal CRUFD operation

Provider responds with an Event message instead
- Header property to indicate this is an event, not a normal payload.
- Body corresponds to the critical fields in a Redfish Event.

After processing Event, MC would send the original request again.

While PLDM supports async messages, not all implementations support it. For instance, it is not practical on I2C.

- The type of eventing needs to be negotiated at initialization.
- Proposal: Negotiate at initialization
  - The Device indicate to the host the types of eventing it supports for this connection as part of initialization.
  - The Host then indicates to the device which one in the list will be used on the connection.
- Part of the initialization handshake could indicate which registries are used by the device.
- Format: PLDM has a format, but since PLDM 2.0 is open then perhaps we need to add the capability of extending/adding a format to accommodate the Redfish eventing format.

Tasks

- Some operations are expected to take longer than the PLDM message time.
  - Example, rebuilding a RAID set.
- Redfish model is there is a task service.
  - In it is a collection of tasks.
  - The client of the MC polls on a unique number or does a GET on the task resource from the MC.
  - When the task completes, there is a place for the response as if it was a normal response.
- Need Task ID.
  - Each Device/Provider can use it's own task IDs.
  - The MC needs to keep track of the task IDs per provider.
  - This is an internal reference number and not the same as that given to the end client via Redfish.
- As part of the response to the operation:
  - We need to indicate this is an async operation and provide the TaskID to the MC
- When the task completes
  - The eventing mechanism can be used to indicate the Task is complete
  - The MC can then ask the Device for the Task status & completion information.
    - This can then be leveraged for polling
    - This may need a unique format.
- Need to accommodate the Delete Task operation.
- Need to look at PLDM FW Update and see if/what can be leveraged.
  - Redfish doesn't have percent complete for tasks, but perhaps this could be leveraged.

Details on Initialization & Decomissioning

Initialization Rules

We need to cover a couple of corner cases

- When a device resets itself and its data
  - The device needs to present itself as the same device so the MC doesn't get confused
    - Provided it isn't fundamentally changed into a different device.
  - PLDM already handles this

- When the MC resets the information it has about a device
  - The device needs to present itself as the same device so the MC doesn't get confused,
    - provided it isn't fundamentally changed into a different device.
  - PLDM already handles this

- Device replacement:
  - Treat this as remove/add?
  - Allow a vendor to try to preserve the information if it so chooses?

- Any other corner cases:

Decommissioning Rules

This can happen a couple of ways:

Note: As far as the spec goes, all of this is a recommendation and it's the MC's problem. This is a use case to ensure that the initialization information supports the MC's ability to perform these functions.

Device no longer addressable

- MC should mark all resources as gone and ensure clients cannot reach them
- After an appropriate period of time (MC's discretion, might want this configurable), the MC would remove any unneeded content (caches payloads, local schema)

Device Replaced Completely

- If this is a different resource, the MC should proceed as above and also perform initial handshake.
- If this is the same make/model:
  - We have some choices to make here

Resource Gone but Device is still there

- During it's periodic poll or request, the MC may find the resource gone or replaced with a different resource.
- The Resource responds with packet indicating it has new resources.
- This should be a format similar to the initial handshake with a list of resource that have been removed as well as resources that have been added.
- MC would garbage collect any cached schema/resource state for the removed resources

Other Corner Cases

FW Update

- During FW update, the schema may change.
  - It may change to a backward incompatible version (1.0 to 2.0)
- We need to figure out what to do in this case
  - Goal would be to preserve as much of the information and settings as possible.
  - Possibly this is a removal and addition.
  - The device would be the best to determine.
- Is this something the spec can be silent on and let implementations determine the proper course?