**Version: 1.0**

**Date: 2019-01-23**

**Author: Deepak Kodihalli**

**Email:** [**dkodihal@in.ibm.com**](mailto:dkodihal@in.ibm.com)

**Platform Level Data Model (PLDM) for File Based I/O**

**Scope**

The scope of this specification is to define the data structures and commands to read/write files between two PLDM termini. Platform specific use-cases are outside the scope of this specification. The contents of the files and details about how the files are stored are outside the scope of this specification. The contents of the file and metadata about the files (for e.g.: file name, file size) are understood by the two PLDM termini, but otherwise are platform/implementation specific. Some examples include:

* Host firmware bootloader reading sections of the host firmware image from a BMC. The host firmware image may have been stored on a NOR flash chip, to which the BMC alone has access.
* Management device firmware storing settings it requires to run the next time on an NVRAM on the BMC, for lack of a large enough persistent storage on the management device.

**Overview**

Before accessing any files, a PLDM requester must obtain information about the set of files that are currently present with the PLDM responder. This may not be the entire set of files that the requester is interested in. The information about the files contains well-known file names (names that are known by the requester and responder in a platform), file sizes and file traits. Such information is contained in tables, and these tables may be updated. Once a requester is aware of a file name and file size, commands to read/write the file may be issued.

**Tables**

*1) File Attribute Table*

This table comprises of metadata for files. Metadata includes the file handle, file name, current file size and traits related to file access. The handles used are unique per file name, and they shouldn’t change for the same name across any two instances of reading this table. The assignment of the handle values is implementation specific. Following is an example of this table:

|  |  |  |  |
| --- | --- | --- | --- |
| **File Handle** | **File Name** | **File Size** | **File Traits** |
| 0x00000000 | GARD | 1024 | 0b0 |
| 0x00000021 | NVRAM | 4096 | 0b100 |
| … |  | … | … |

Following is the structure of the file attribute table:

|  |  |  |
| --- | --- | --- |
| **Byte** | **Type** | **Field** |
| 0:3 | uint32 | **FileHandle[0]**  The handle for the first file |
| 4:5 | uint16 | **FileNameLength[0]**  The length of the first file name in bytes |
| Variable |  | **FileName[0]**  The first file name |
|  | uint32 | **FileSize[0]**  The size of the first file in bytes |
|  | bitfield32 | **FileTraits[0]**  Traits for the first file  [31:3] – Reserved  [2] – Preserve across firmware upgrades  [1] - ReadWrite  [0] – ReadOnly |
|  | uint32 | **FileHandle[1]**  The handle for the second file |
|  | uint16 | **FileNameLength[1]**  The length of the second file name in bytes |
|  |  | **FileName[1]**  The second file name |
|  | uint32 | **FileSize[1]**  The length of the second file in bytes |
|  | bitfield32 | **FileTraits[1]**  Traits for the second file  [31:3] - Reserved  [2] - Preserve across firmware upgrades  [1] - ReadWrite  [0] - ReadOnly |
| … | … | **…** |
| Variable | … | **Pad**  0 to 3 number of pad bytes. The value stored in each pad byte is 0x00.  The transmitter can compute the number of pad bytes from the FileAttributeTable by using the following algorithm:  Let L be the total number of bytes in the FileAttributeTable excluding the pad and the integrity checksum.  if (L modulo 4 == 0) then NumPadBytes = 0; else NumPadBytes = 4 – L modulo 4;  The receiver can compute the number of pad bytes from the FileAttributeTable by using the following algorithm. In the algorithm, the receiver parses FileAttributeTable until the remaining bytes are less than 8. When it reaches that stage, the remaining bytes contain the  pad bytes and four bytes of data integrity checksum. Let L be the total number of bytes in the FileAttributeTable including the pad and the  integrity checksum.  RemBytes = L;  i = 0;  while (RemBytes >= 8)  {  Process the ith string in the table;  RemBytes = RemBytes - 4 - String i Length;  i = i+1;  }  NumPadBytes = RemBytes modulo 4; |
|  | uint32 | **FileAttributeTableIntegrityChecksum**  Integrity checksum on the FileAttributeTable shown above including the pad bytes (if any). For this specification, the CRC-32 algorithm with the polynomial x^32 + x^26 + x^23 + x^22 + x^16 + x^12 + x^11 + x^10 + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1 (same as the one used by IEEE 802.3) must be used for the integrity checksum computation. The CRC computation involves processing a byte at a time with the least significant bit first. |

*3) OEM File Attribute Table*

This table lets OEMs add their own file traits.

|  |  |  |
| --- | --- | --- |
| **Byte** | **Type** | **Field** |
| 0:3 | uint32 | **FileHandle[0]**  The handle for the first file |
| 4:7 | bitfield32 | **FileTraits[0]**  Traits for the first file |
|  | uint32 | **FileHandle[1]**  The handle for the second file |
|  | bitfield32 | **FileTraits[1]**  Traits for the second file |
| … | … | **…** |
| Variable | … | **Pad**  0 to 3 number of pad bytes. The value stored in each pad byte is 0x00.  The transmitter can compute the number of pad bytes from the OEMFileAttributeTable by using the following algorithm:  Let L be the total number of bytes in the OEMFileAttributeTable excluding the pad and the integrity checksum.  if (L modulo 4 == 0) then NumPadBytes = 0; else NumPadBytes = 4 – L modulo 4;  The receiver can compute the number of pad bytes from the OEMFileAttributeTable by using the following algorithm. In the algorithm, the receiver parses OEMFileAttributeTable until the remaining bytes are less than 8. When it reaches that stage, the remaining bytes contain the pad bytes and four bytes of data integrity checksum. Let L be the total number of bytes in the OEMFileAttributeTable including the pad and the integrity checksum.  RemBytes = L;  i = 0;  while (RemBytes >= 8)  {  Process the ith string in the table;  RemBytes = RemBytes - 4 - String i Length;  i = i+1;  }  NumPadBytes = RemBytes modulo 4; |
|  | uint32 | **OEMFileAttributeTableIntegrityChecksum**  Integrity checksum on the OEMFileAttributeTable shown above including the pad bytes (if any). For this specification, the CRC-32 algorithm with the polynomial x^32 + x^26 + x^23 + x^22 + x^16 + x^12 + x^11 + x^10 + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1 (same as the one used by IEEE 802.3) must be used for the integrity checksum computation. The CRC computation involves processing a byte at a time with the least significant bit first. |

**Commands**

*1) GetFileTable*

This command is used to retrieve a File table. Multipart transfers are possible.

|  |  |  |
| --- | --- | --- |
| **Byte** | **Type** | **Request Data** |
| 0:3 | uint32 | **DataTransferHandle**  A handle that is used to identify a File table transfer. This handle is ignored by the responder when the TransferOperationFlag is set to GetFirstPart. |
| 4 | enum8 | **TransferOperationFlag**  The transfer operation flag that indicates whether this is the start of a multipart transfer  Possible values: {GetNextPart=0x00, GetFirstPart=0x01} |
| 5 | enum8 | **TableType**  Indicates what table is being transferred  Possible values:  {  FileAttributeTable=0x0,  OEMFileAttributeTable=0x1  } |
| **Byte** | **Type** | **Request Data** |
| 0 | enum8 | CompletionCode  Possible values:  {  PLDM\_BASE\_CODES,  INVALID\_DATA\_TRANSFER\_HANDLE=0x80,  INVALID\_TRANSFER\_OPERATION\_FLAG=0x81,  FILE\_TABLE\_UNAVAILABLE=0x83,  INVALID\_FILE\_TABLE\_DATA\_INTEGRITY\_CHECK=0x84,  INVALID\_FILE\_TABLE\_TYPE=0x85  } |
| 1:4 | uint32 | **NextDataTransferHandle**  A handle that is used to identify the next portion of the transfer |
| 5 | enum8 | **TransferFlag**  The transfer flag that indicates what part of the transfer this response represents  Possible values: {Start = 0x1, Middle = 0x2, End = 0x4, StartAndEnd = 0x5} |
| Variable | - | **TableData**  Table type specific data |

*2) SetFileTable*

This command is used to provide a new file table. This should replace an already existing file table.

|  |  |  |
| --- | --- | --- |
| **Byte** | **Type** | **Request Data** |
| 0:3 | uint32 | **DataTransferHandle**  A handle that is used to identify a File table transfer. This handle is ignored by the responder when the TransferFlag is set to Start or StartAndEnd. |
| 4 | enum8 | **TransferFlag**  The transfer flag that indicates what part of the transfer this request represents  Possible values: {Start = 0x1, Middle = 0x2, End = 0x4, StartAndEnd = 0x5} |
| 5 | enum8 | **TableType**  Indicates what table is being transferred  Possible values:  { FileAttributeTable=0x0, OEMFileAttributeTable=0x1} |
| Variable | - | **TableData**  Table type specific data. |
| **Byte** | **Type** | **Response Data** |
| 0 | enum8 | CompletionCode  Possible values:  {  PLDM\_BASE\_CODES,  INVALID\_DATA\_TRANSFER\_HANDLE=0x80,  INVALID\_TRANSFER\_FLAG=0x82,  INVALID\_FILE\_TABLE\_DATA\_INTEGRITY\_CHECK=0x84,  INVALID\_FILE\_TABLE\_TYPE=0x85  } |
| 1:4 | uint32 | **NextDataTransferHandle**  A handle that is used to identify the next portion of the transfer |

*3) UpdateFileTable*

This command is used to update a file table. This means adding new entries, updating existing entries, or both.

|  |  |  |
| --- | --- | --- |
| **Byte** | **Type** | **Request Data** |
| 0:3 | uint32 | **DataTransferHandle**  A handle that is used to identify a File table transfer. This handle is ignored by the responder when the TransferFlag is set to Start or StartAndEnd. |
| 4 | enum8 | **TransferFlag**  The transfer flag that indicates what part of the transfer this request represents  Possible values: {Start = 0x1, Middle = 0x2, End = 0x4, StartAndEnd = 0x5} |
| 5 | enum8 | **TableType**  Indicates what table is being transferred  Possible values:  { FileAttributeTable=0x0, OEMFileAttributeTable=0x1} |
| Variable | - | **TableData**  Table type specific data. |
| **Byte** | **Type** | **Response Data** |
| 0 | enum8 | CompletionCode  Possible values:  {  PLDM\_BASE\_CODES,  INVALID\_DATA\_TRANSFER\_HANDLE=0x80,  INVALID\_TRANSFER\_FLAG=0x82,  INVALID\_FILE\_TABLE\_DATA\_INTEGRITY\_CHECK=0x84,  INVALID\_FILE\_TABLE\_TYPE=0x85  } |
| 1:4 | uint32 | **NextDataTransferHandle**  A handle that is used to identify the next portion of the transfer |

*4) ReadFile*

This command is used to read a file.

|  |  |  |
| --- | --- | --- |
| **Byte** | **Type** | **Request Data** |
| 0:3 | uint32 | **FileNameHandle**  A handle to the file |
| 4:7 | uint32 | **Offset**  Offset to the file at which the read should begin |
| 8:11 | uint32 | **Length**  Number of bytes to be read |
| **Byte** | **Type** | **Response Data** |
| 0 | enum8 | **CompletionCode**  value:  {  PLDM\_BASE\_CODES,  INVALID\_FILE\_HANDLE=0x86,  DATA\_OUT\_OF\_RANGE=0x87  }  If request offset > file size, then DATA\_OUT\_OF\_RANGE shall be returned. |
| 1:4 | uint32 | **Length**  Number of bytes read. This could be less than what the requester asked for. |
| Variable | - | **FileData**  File data starting from request offset to (request offset + response length – 1) |

*5) WriteFile*

This command is used to write to a file. If the file has to be created newly, then an UpdateFileTable is expected before this command is sent. Appends to existing files will have to result in changes to the file size property in the file attribute table.

|  |  |  |
| --- | --- | --- |
| **Byte** | **Type** | **Request Data** |
| 0:3 | uint32 | **FileNameHandle**  A handle to the file |
| 4:7 | uint32 | **Offset**  Offset in the file at which the write should begin |
| 8:11 | uint32 | **Length**  Number of bytes to be written |
| Variable | - | **FileData**  File data starting from request offset to (request offset + request length – 1) |
| **Byte** | **Type** | **Response Data** |
| 0 | enum8 | **CompletionCode**  value:  {  PLDM\_BASE\_CODES,  INVALID\_FILE\_HANDLE=0x86,  READ\_ONLY=0x88  }  If request offset + response length > current file size, then file size should be updated in the file attribute table |
| 1:4 | uint32 | **Length**  Number of bytes written. This could be less than what the requester asked for. |

*6) ReadFileIntoMemory*

This command is used to read a file. The read content is written to a memory region, an address to which is in the request. This is as opposed to sending the file content as a response payload.

|  |  |  |
| --- | --- | --- |
| **Byte** | **Type** | **Request Data** |
| 0:3 | uint32 | **FileNameHandle**  A handle to the file |
| 4:7 | uint32 | **Offset**  Offset to the file at which the read should begin |
| 8:11 | uint32 | **Length**  Number of bytes to be read |
| 12:19 | uint64 | **Address**  Memory address where the file content has to be written to |
| **Byte** | **Type** | **Response Data** |
| 0 | enum8 | **CompletionCode**  value:  {  PLDM\_BASE\_CODES,  INVALID\_FILE\_HANDLE=0x86,  DATA\_OUT\_OF\_RANGE=0x87  }  If request offset > file size, then DATA\_OUT\_OF\_RANGE shall be returned. |
| 1:4 | uint32 | **Length**  Number of bytes read. This could be less than what the requester asked for. |

*7) WriteFileFromMemory*

This command is used to write to a file. If the file has to be created newly, then an UpdateFileTable is expected before this command is sent. Appends to existing files will have to result in changes to the file size property in the file attribute table.

This version of the command expects the responder to read the file content from a memory location pointed to by the request, as opposed to sending the file content as a request payload.

|  |  |  |
| --- | --- | --- |
| **Byte** | **Type** | **Request Data** |
| 0:3 | uint32 | **FileNameHandle**  A handle to the file |
| 4:7 | uint32 | **Offset**  Offset in the file at which the write should begin |
| 8:11 | uint32 | **Length**  Number of bytes to be written |
| 12:19 | uint64 | **Address**  Memory address where the file content has to be read from |
| **Byte** | **Type** | **Response Data** |
| 0 | enum8 | **CompletionCode**  value:  {  PLDM\_BASE\_CODES,  INVALID\_FILE\_HANDLE=0x86,  READ\_ONLY=0x88  }  If request offset + response length > current file size, then file size should be updated in the file attribute table |
| 1:4 | uint32 | **Length**  Number of bytes written. This could be less than what the requester asked for. |

*8) ReadFileByTypeIntoMemory*

This command is used to read a file, by type and handle. The read content is written to a memory region, an address to which is in the request.

|  |  |  |
| --- | --- | --- |
| **Byte** | **Type** | **Request Data** |
| 0:1 | enum16 | **FileType**  Type of the file |
| 2:5 | uint32 | **FileHandle**  A handle to the file |
| 6:9 | uint32 | **Offset**  Offset to the file at which the read should begin |
| 10:13 | uint32 | **Length**  Number of bytes to be read |
| 14:21 | uint64 | **Address**  Memory address where the file content has to be written to |
| **Byte** | **Type** | **Response Data** |
| 0 | enum8 | **CompletionCode**  value:  {  PLDM\_BASE\_CODES,  INVALID\_FILE\_HANDLE=0x86,  DATA\_OUT\_OF\_RANGE=0x87,  INVALID\_FILE\_TYPE=0x89  }  If request offset > file size, then DATA\_OUT\_OF\_RANGE shall be returned. |
| 1:4 | uint32 | **Length**  Number of bytes read. This could be less than what the requester asked for. |

*9) WriteFileByTypeFromMemory*

This command is used to write to a file, by type. A file handle of 0xFFFFFFFF would indicate to not store the file metadata in the file table.

This version of the command expects the responder to read the file content from a memory location pointed to by the request.

|  |  |  |
| --- | --- | --- |
| **Byte** | **Type** | **Request Data** |
| 0:1 | enum16 | **FileType**  Type of the file |
| 2:5 | uint32 | **FileHandle**  A handle to the file |
| 6:9 | uint32 | **Offset**  Offset in the file at which the write should begin |
| 10:13 | uint32 | **Length**  Number of bytes to be written |
| 14:21 | uint64 | **Address**  Memory address where the file content has to be written to |
| **Byte** | **Type** | **Response Data** |
| 0 | enum8 | **CompletionCode**  value:  {  PLDM\_BASE\_CODES,  INVALID\_FILE\_HANDLE=0x86,  READ\_ONLY=0x88,  INVALID\_FILE\_TYPE=0x89  } |
| 1:4 | uint32 | **Length**  Number of bytes written. This could be less than what the requester asked for. |

*10) NewFileAvailable*

This command is used to denote that a new file is available.

|  |  |  |
| --- | --- | --- |
| **Byte** | **Type** | **Request Data** |
| 0:1 | enum16 | **FileType**  Type of the file |
| 2:5 | uint32 | **FileHandle**  A handle to the file |
| 6:13 | uint64 | **Length**  Size of the file |
| **Byte** | **Type** | **Response Data** |
| 0 | enum8 | **CompletionCode**  value:  {  PLDM\_BASE\_CODES,  INVALID\_FILE\_HANDLE=0x86,  INVALID\_FILE\_TYPE=0x89  } |

*11) ReadFileByType*

This command is used to read a file, specified by type and handle.

|  |  |  |
| --- | --- | --- |
| **Byte** | **Type** | **Request Data** |
| 0:1 | enum16 | **FileType**  Type of the file |
| 2:5 | uint32 | **FileNameHandle**  A handle to the file |
| 6:9 | uint32 | **Offset**  Offset to the file at which the read should begin |
| 10:13 | uint32 | **Length**  Number of bytes to be read |
| **Byte** | **Type** | **Response Data** |
| 0 | enum8 | **CompletionCode**  value:  {  PLDM\_BASE\_CODES,  INVALID\_FILE\_HANDLE=0x86,  DATA\_OUT\_OF\_RANGE=0x87,  INVALID\_FILE\_TYPE=0x89  }  If request offset > file size, then DATA\_OUT\_OF\_RANGE shall be returned. |
| 1:4 | uint32 | **Length**  Number of bytes read. This could be less than what the requester asked for. |
| Variable | - | **FileData**  File data starting from request offset to (request offset + response length – 1) |

*12) WriteFileByType*

This command is used to write to a file, specified by type and handle.

|  |  |  |
| --- | --- | --- |
| **Byte** | **Type** | **Request Data** |
| 0:1 | enum16 | **FileType**  Type of the file. |
| 2:5 | uint32 | **FileNameHandle**  A handle to the file |
| 6:9 | uint32 | **Offset**  Offset in the file at which the write should begin |
| 10:13 | uint32 | **Length**  Number of bytes to be written |
| Variable | - | **FileData**  File data starting from request offset to (request offset + request length – 1) |
| **Byte** | **Type** | **Response Data** |
| 0 | enum8 | **CompletionCode**  value:  {  PLDM\_BASE\_CODES,  INVALID\_FILE\_HANDLE=0x86,  READ\_ONLY=0x88,  INVALID\_FILE\_TYPE=0x89  }  If request offset + response length > current file size, then file size should be updated in the file attribute table |
| 1:4 | uint32 | **Length**  Number of bytes written. This could be less than what the requester asked for. |

*13) FileAck*

This command is used to denote that a file that was transferred has been processed.

|  |  |  |
| --- | --- | --- |
| **Byte** | **Type** | **Request Data** |
| 0:1 | enum16 | **FileType**  Type of the file |
| 2:5 | uint32 | **FileHandle**  A handle to the file |
| 6 | enum8 | **FileStatus**  **Status of the file transfer**  **value:**  {  PLDM\_BASE\_CODES,  ERROR\_FILE\_DISCARDED=0x8A,  FULL\_FILE\_DISCARDED=0x8B  AUTHORITY\_VALIDATION\_ERROR=0x8C  } |
| **Byte** | **Type** | **Response Data** |
| 0 | enum8 | **CompletionCode**  value:  {  PLDM\_BASE\_CODES,  INVALID\_FILE\_HANDLE=0x86,  INVALID\_FILE\_TYPE=0x89,  } |

=========================================================

Proposal: *NewFileAvailableWithMetadata* / *FileAckWithMetadata*

=========================================================

14) *NewFileAvailableWithMetadata*

This command is used to a new file is available, and that the file has type-specific metadata. The type-specific metadata is described later in this document.

|  |  |  |
| --- | --- | --- |
| **Byte** | **Type** | **Request Data** |
| 0:1 | enum16 | **FileType**  Type of the file |
| 2:5 | uint32 | **FileHandle**  A handle to the file |
| 6:13 | uint64 | **Length**  Size of the file |
| 14:17 | uint32 | **File Metadata #1** |
| 18:21 | uint32 | **File Metadata #2** |
| 22:25 | uint32 | **File Metadata #3** |
| 26:29 | uint32 | **File Metadata #4** |
| **Byte** | **Type** | **Response Data** |
| 0 | enum8 | **CompletionCode**  value:  {  PLDM\_BASE\_CODES,  INVALID\_FILE\_HANDLE=0x86,  INVALID\_FILE\_TYPE=0x89  } |

15) *FileAckWithMetadata*

This command is used to denote that a file that was transferred has been processed, and that file metadata should be added or updated. The type-specific metadata is described later in this document.

|  |  |  |
| --- | --- | --- |
| **Byte** | **Type** | **Request Data** |
| 0:1 | enum16 | **FileType**  Type of the file |
| 2:5 | uint32 | **FileHandle**  A handle to the file |
| 6 | enum8 | **FileStatus**  **Status of the file transfer**  **value:**  {  PLDM\_BASE\_CODES,  ERROR\_FILE\_DISCARDED=0x8A,  FULL\_FILE\_DISCARDED=0x8B  AUTHORITY\_VALIDATION\_ERROR=0x8C  } |
| 10:13 | uint32 | **File Metadata #1** |
| 14:17 | uint32 | **File Metadata #2** |
| 18:21 | uint32 | **File Metadata #3** |
| 22:25 | uint32 | **File Metadata #4** |
| **Byte** | **Type** | **Response Data** |
| 0 | enum8 | **CompletionCode**  value:  {  PLDM\_BASE\_CODES,  INVALID\_FILE\_HANDLE=0x86,  INVALID\_FILE\_TYPE=0x89,  } |

=========================================================

**Types of Files**

enum Type

{

ERROR\_LOG = 0x0000,

LID\_PERM = 0x0001,

LID\_TEMP = 0x0002,

DUMP = 0x0003,

CERT\_SIGNING\_REQUEST = 0x0004,

SIGNED\_CERT = 0x0005,

ROOT\_CERT = 0x0006,

FILE\_TYPE\_MARKER\_LID = 0x0007,

RESOURCE\_DUMP\_PARMS = 0x0008,

RESOURCE\_DUMP = 0x0009,

PROGRESS\_SRC = 0x000A

ADJUNCT\_DUMP = 0x000B, // LPA

DEVICE\_DUMP = 0x000C, // HMS

COD\_LICENSE\_KEY = 0x000D,

COD\_LICENSED\_RESOURCES = 0x000E,

UNDEFINED = 0xFFFF,

}

*File handles for specific types:*

ERROR\_LOG : error id (EID)

LID\_PERM/LID\_TEMP : LID id

DUMP : Dump id

CERT\_SIGNING\_REQUEST/ SIGNED\_CERT: file handle is set by bmcweb and used in all subsequent transactions

ROOT\_CERT: filehandle 0x00000000

Format of the Resource Dump Parameters :

<Length of the VSP String><VSP String><Length of the Password><Password>

**File Metadata & Behaviors**

Certain file types may have type-specific metadata associated with them, as denoted below. If a field is unused, or the value is unknown by a terminus and should be filled in later by another terminus, the value 0xFFFFFFFF shall be used.

File Metadata for RESOURCE\_DUMP\_PARMS = 0x0008, RESOURCE\_DUMP = 0x0009, ADJUNCT\_DUMP = 0x000B, and DEVICE\_DUMP = 0x000C

|  |  |
| --- | --- |
| **File Metadata #1** | Resource Dump Correlation Token (PLID)   * When passing the initial RESOURCE\_DUMP\_PARMS file to PHYP, BMC uses the traditional *NewFileAvailable* command (includes no metadata). * When PHYP acknowledges the RESOURCE\_DUMP\_PARMS file using *FileAckWithMetadata,* PHYP will select a token for correlating the dump request to the dump file via whatever method it chooses (traditionally PLID). The selected token will be placed in this field. * When PHYP responds with a RESOURCE\_DUMP, ADJUNCT\_DUMP, or DEVICE\_DUMP via *NewFileAvailableWithMetadata*, it will set this field to match the token selected in the previous step. * When BMC has processed the dump file, it will acknowledge with *FileAckWithMetadata, and will maintain the token PHYP selected in previous steps.* |
| **File Metadata #2** | Resource Dump Parms Status Code   * When passing the initial RESOURCE\_DUMP\_PARMS file to PHYP, BMC uses the traditional *NewFileAvailable* command (includes no metadata). * When PHYP acknowledges the RESOURCE\_DUMP\_PARMS file using *FileAckWithMetadata, PHYP will fill in this field with a status code for the request. Quick Pass at status codes*   + *0 == Success (Security Validated)*   + *1 == ACF File Invalid*   + *2 == Password Invalid*   + *3 == Permission Denied*   + *4== Resource Selector Invalid* * *This field is not used for RESOURCE\_DUMP = 0x0009, ADJUNCT\_DUMP = 0x000B, OR DEVICE\_DUMP = 0x000C* |
| **File Metadata #3** | Resource Selector Invalid Detailed Status Code   * When Resource Dump Parms Status Code is set to Resource Selector Invalid, PHYP will fill in this field with a detailed status code.   + 1== InvalidParmLength   + 2== UnknownParmVersion   + 4== UnknownClientType   + 5== InvalidCommandString   + 6== InvalidState   + 7== Busy   + 8== UnknownDump   + 9== InvalidOffset   + 10== EndOfFileReached * For all other Resource Dump Parms Status Codes, this value is Unused, set to 0xFFFFFFFF |
| **File Metadata #4** | Unused, set to 0xFFFFFFFF |

File Metadata for COD\_LICENSE\_KEY = 0x000D

|  |  |
| --- | --- |
| **File Metadata #1** | License Processing Result   * When passing the initial COD\_LICENSE\_KEY file to PHYP via *NewFileAvailable/NewFileAvailableWithMetadata*, BMC will set this to 0xFFFFFFFF == unknown * When PHYP acknowledges the COD\_LICENSE\_KEY file using *FileAckWithMetadata,* PHYP will set one of the following values that may be mapped to a Redfish construct (message?):   + 0x00000000 : The license was installed successfully.   + 0x00000001 : Invalid License – the license is not valid   + 0x00000002 : Incorrect System – the license is valid, but not for the system it was installed on.   + 0x00000003 : Incorrect Sequence – the license is valid on this system, but is in the incorrect sequence (eg, was entered more than once).   + A couple others, get details from Paul Olsen here... |
| **File Metadata #2** | Unused, set to 0xFFFFFFFF |
| **File Metadata #3** | Unused, set to 0xFFFFFFFF |
| **File Metadata #4** | Unused, set to 0xFFFFFFFF |

**All other file types should have all metadata fields set to 0xFFFFFFFF.**