
# Data Object Manager Software High Level Design

On-Ramp Wireless Incorporated
10920 Via Frontera, Suite 200
San Diego, CA 92127
U.S.A.



Data Object Manager Software High Level Design

013-xxxx-00 Rev. A

August 12, 20144

# Contents

# Figures



# Tables

**No table of figures entries found.**

# Revision History

| Revision | Release Date | Change Description |
|---|---|---|
| A | Aug 5, 2014 | Initial release. |

# 1 Introduction

## 1.1 Purpose

This document describes the software implementation and methods that make up the Data Object Manager (DOM) software component used to manage and provide wear-leveling for a user-specified Binary Data Object in a non-memory mapped memory device (e.g., external SPI-Flash). The software design is primarily intended for On-Ramp Wireless' AMI meter solutions (e.g., Raptor, Ptero, etc), but the design can also easily be ported to other embedded platforms that do not have a File System component included with their run-time environment.

## 1.2 Scope

This high level design (HLD) document identifies each of the functional software blocks used on an application host processor which are responsible for configuring and managing all accesses to a user-defined Binary Data Object (DO). Also discussed is the underlying format of the meta-data stored in NVM used by the DOM for committing and getting said Data Object.

The actual implementation of the hardware drivers responsible for saving or reading data to and from a external memory device are, however, beyond the scope of this document. The Data Object Manager is implemented such that the details of managing and external memory are abstracted out so that it can be used for a number of vendor-agnostic devices.

## 1.3 References

The Data Object Manager and its underlying implementation is a strictly a proprietary ORW software design and is not dependent on any external software library and/or third-party software implementation. As such, there are no external references.

# 2 DOM Architecture Overview

## 2.1 Software Design Requirements

The following software design requirements are considered for the Data Object Manager (DOM) software component:

- The AMI application will need use of a finite set of frequently-updated data objects (DO) that need to be located in persistent memory and can be quickly restored following exception system resets and/or periods of power outages.
- The frequency of application updates to these persistent, frequently-updated data objects (DO) will, over a 20-year lifetime of the MCM platform, will exceed the rated flash-erase cycle of a typical Flash part without some sort of wear-leveling management.
- In addition to wear-leveling, the AMI application will require some sort of “pseudo-dynamic” allocation of flash management Data Blocks (DB) for different sized and number of data-objects used by the application. The application will be responsible for determining the total amount of memory necessary for a specified data object (DO) that will meet the desired rated flash-erase cycle for the MCM platform.
- There will be a “fixed” NVM size reserved for use by the sum total of “pseudo-dynamic” allocated flash management Data Blocks (DB).
- There will be only be a single active copy of a given data object (DO) actively used/accessed by the AMI application.
- The underling NVM Data Object Manager (DOM) is responsible for performing all of the underlying bookkeeping necessary to keep track and commit the active copy of a given data object into physical NVM.
- The NVM Data Object Manager (DOM) will also need to support partial fragment reads and writes to a given data object (DO), although this is primarily intended for large binary objects (e.g., objects that span multiple data sectors in physical memory).
- There will be an error detection and subsequent recovery mechanism to properly manage unexpected reset conditions (e.g. bus fault, hardware fault, manual reset line toggle, etc) that may occur during the update to NVM of a given Data Object (DO) and/or its associated meta-data.
- The NVM Data Object Manager (DOM) will also support an optional authentication and/or encryption of data saved in physical memory.
- Data Object (DO) initialization and Data Object Management (DOM) meta-data will need to be preserved across software upgrades. If the format of the DOM meta-data changes between software upgrades, then the DOM initialization process will recovery and convert the last active Data Object to use the new meta-data format.

- Data Object Management (DOM) initialization and/or large-object allocation may involve time-consuming bulk erases – as such, these two methods will support both blocking and non-blocking/re-entrant bulk erase methods.

Listed are the assumptions and limitations of the proposed implementation:

- It is *not* guaranteed that data objects stored in physical NVM can be directly memory accessed (e.g., external SPI Flash). As a result, the active Data Object (DO) is always located in RAM and passed as a reference into the Data Object Manager (i.e., it is the responsibility of the application layer to allocated the area of RAM memory for it).
- The Data Blocks (DB) are assumed to be uniform/minimum size elements within NVM that can be individually erased – e.g., flash sectors blocks. This simplifies the overall Data Object Management (DOM) meta-data record keeping.
- Since Data Object (DO) management involves the use of the NVM PAL and flash programming drivers which will require callbacks and/or status polling, it is not safe to use any of these methods in interrupt context.
- There are currently no statistics and/or counters to report how frequently the Data Blocks within NVM are updated and/or erased – this type of information, to be of use, will in turn required its own persistent memory assets to be of any use. It is the responsibility of the user application to properly allocate enough memory for a given Data Object to meet the expected wear-leveling requirements over the expected life-span of the embedded device.

## 2.2 Application Overview

The Data Object Manager (DOM) software component is implemented in a single module in an application host's Hardware Abstraction Layer – in the case of the EMCM application, this is the PAL. It serves as the interface between the application layer and the NVM drivers for all Binary Data Objects that are registered during initialization time.

It is the responsibility of the application layer to allocate RAM memory for the working/active copy of a given Data Object (DO). All application layer read/write accesses are to this cached working/active copy. It is also the responsibility of the application layer to determine when updates to the cached working/active copy are required to be written into persistent NVM through use of the Data Object Management software component (see next section for a list of the provided APIs).

For large Binary Data Objects on applications that have limited memory resources, the cached data can be done in manageable "fragments" which can be refreshed or committed into the larger data object in persistent NVM as needed.

All of the variables necessary for Data Object management and mapping to physical NVM are private in scope and are only accessed by the DOM software component.

All accesses to physical NVM by the DOM software component are done through the existing NVM PAL driver – for design simplicity, most calls into the NVM PAL will use blocking methods – the one exception are accesses that involve bulk erases over multiple sectors/blocks. In addition, the NVM PAL module will also be responsible for determining which memory block in physical NVM is allocated for use by the Data Object Manager (with APIs to retrieve this information).

The following diagram illustrates how the Data Object Manager interfaces with a typical application:

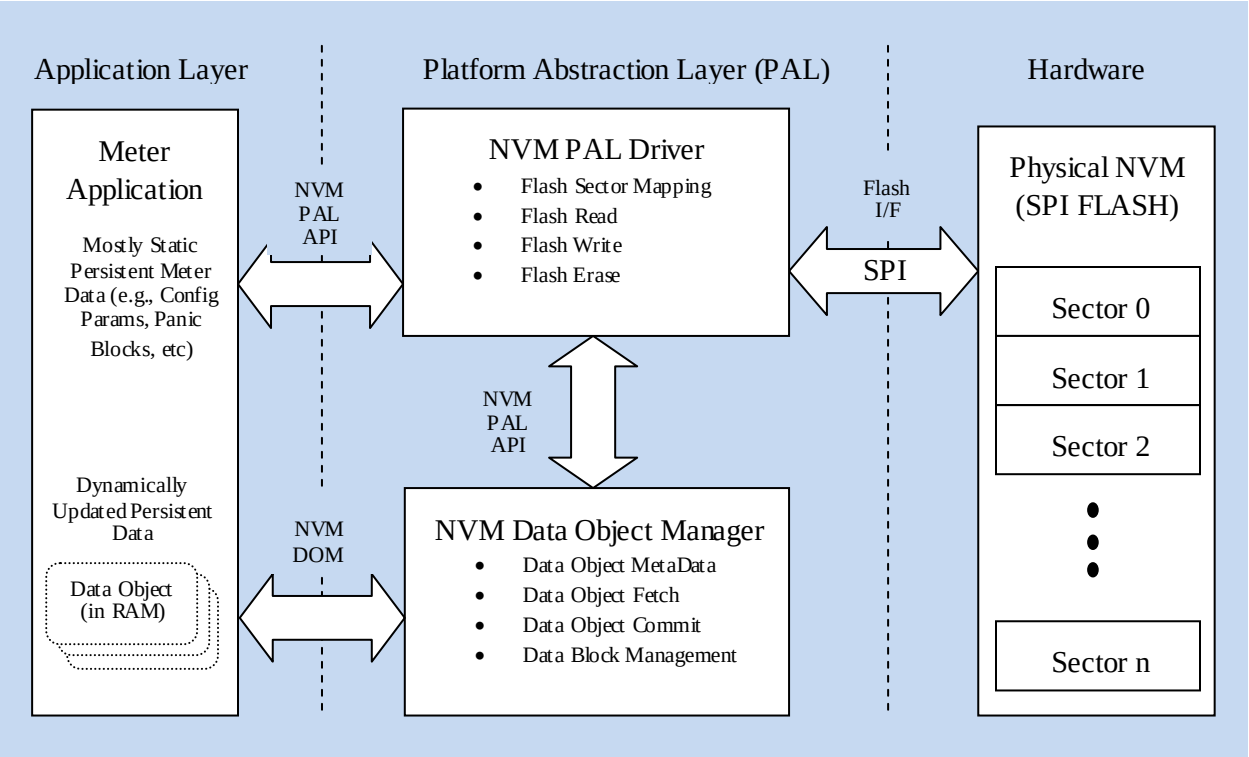


**Figure 1. Data Object Manager Application Overview**

# 2.3 Run-Time Environment

The Data Object Manager is designed to be accessed from a single level of process execution. In systems with an RTOS, this should be a single run-time executable or thread. Accessing the DOM through an interrupt context is discouraged. Accesses to the PAL should be done at an equal or higher level context.

# 2.4 DOM States

The Data Object Manager is only concerned with two states of operation:

- **Not Initialized –** The default state of DOM when the application comes out of reset *or* when an access to Persistent NVM fails to read or update DOM meta-data fails.
- **Initialized –** Once the DOM internal management variables and Persistent NVM meta-data has been confirmed and/or programmed, the DOM is ready for use by the application.

DOM state tracking is done on a Data-Block basis – it may be possible to access and/or alter user-data in Data Blocks while the DOM may be performing a re-entrant cleanup of later (and larger Data Blocks).

It is expected that once the application has entered its main processing loop, all of the Data Blocks with be in the Initialized state and prepared to process any user requests.

# 2.5 DOM Memory Resources

The Data Object Manager itself is implemented to use the minimal amount of Code, RAM, and NVM meta-data resources. The following memory estimates are for GCC generated code executing on an ARM Cortex-M3/M4

## 2.5.1 Program FLASH

The PAL DOM implementation currently uses the following program flash space for:

- **Code Space**: ~2500 bytes
- **Read-Only and Initialized Data**: 16 bytes per Data Object Descriptor

NOTE: This program flash usage does not take into consideration the NVM PAL drivers nor any library routines it may call (e.g., memory copy function calls).

## 2.5.2 SRAM

The PAL DOM implementation current uses on-target RAM space for:

- **Data Buffers and tracking variables**: ~400 bytes plus 28 bytes per Data Object Descriptor

## 2.5.3 NVM

As discussed in the next section, each Data Block in NVM that is managed by DOM will have some meta-data overhead required for the management of Application Binary Data Objects. The size of this meta data scales based on the number of Data Objects stored within a Data Block/Sector as follows:

- **NVM Meta-Data Overhead per Data Block/Sector**: 24 bytes plus 1 byte for every 8 Data Objects in a Block/Sector.

In addition, because the underlying management of Data Objects in NVM is block/sector based, there will be some unused NVM memory between the useable Data Object memory and the next block/sector boundary. Because the Binary Data Objects are dynamically assigned/declared at run-time, this unused memory is difficult to quantify (but can be no more than ½ X sector size).

# 3 DOM Functional Components

## 3.1 Data Object Initialization

The user-application has full control and responsibility for how the Data Object Manager manages application data in NVM resources it is configured to control in coordination with the NVM PAL drivers. It is assumed that the NVM PAL has been coded to provide a set of drivers/methods for use by DOM which then provides access to contiguous range of physical NVM. With some work, the DOM can be made to work with multiple/non-contiguous "banks" of physical NVM – but that is a topic that is beyond the scope of the current implementation.

To Initialize DOM, the application will need to set up a list of Data Object descriptors – each descriptor contains the following elements:

- **Descriptor String** – used to uniquely identify a given Data Object – the max size of the descriptor string is currently 8 characters.
- **Data Object Size** – The specified size of the user application binary data object, in bytes. For variable length data, the application will need to specified the max allowed size.
- **Allocated NVM Memory** – To implement wear-leveling, the user application will need to need to specify the total memory in physical NVM, in bytes, to allocate for the current Data object – usual a multiple of the Data Object Size. The application will need to take into consideration the frequency of updates to said data object, the expected lifetime of the product, as well as the erase/write cycles of physical memory. NOTE: The actual NVM memory allocated by DOM may be greater than the specified size as internal Data Object memory allocation rounds up to the nearest block/sector boundary.

The descriptor list passed into the Data Object Manager is terminated with a descriptor that has been zero-initialized.

The Data Object Manager will use this descriptor list to set up all of the Data Object Management internal tracking variables (see next section). In addition, it will check physical NVM to see if the DOM meta-data markers are set up in each Data Block based on the specified descriptor list – if the meta-data is not present or is invalid, then the DOM Init process will perform the erase and write cycles necessary to clear physical NVM and write the correct DOM meta-data.

Because the initialization process can involve a significant amount of bulk erases (and therefore time), the Init process can be done using the following methods:

- **Blocking** – Where the initialization process will block the current execution context until completed.
- **Non-Blocking** – Where the initialization process will allow the current execution context to continue while scheduling all NVM management to be done using re-entrant callbacks and the run-time executable provided by the NVM PAL (if implemented to support re-entrant processing – which is the case on EMCM).

The user application also has the option to skip NVM management of selected Data Objects if it can be guaranteed that the data has been previously setup by the firmware (or programmed via some external method). However, this selective form of initialization should be used with caution – if the NVM meta-data has not been properly setup and the DOM APIs are invoked for such a Data Object, unpredictable/improper programming requests with the NVM PAL will ensue.

## 3.2 Data Object Management

Overall management of each Data Object (DO) and its offset location in Physical NVM in the main responsibility of the Data Object Manager – this, in turn, is managed through a block of meta-data located in RAM and is set-up through the use of the DOM Initialization API. This Data Object Management (DOM) meta-data essentially serves to provide a mapping mechanism of the cached copy of a given Data Object to its physical location offset in NVM as the following figure illustrates for a typical use case where multiple Binary Data Objects can fit into a single NVM Data Block/Sector):

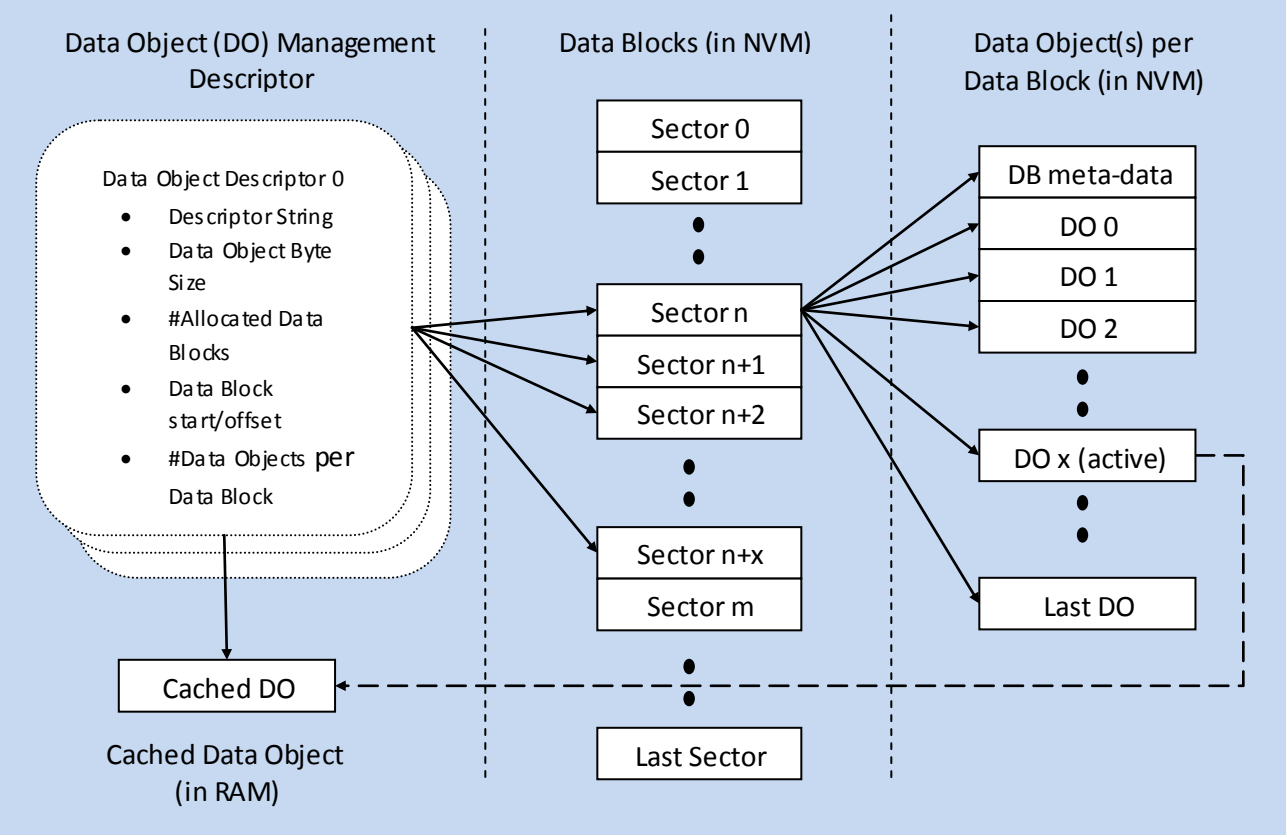


**Figure 2. Data Object Management (Data Object size < Data Block)**

The Data Object Management meta-data is a chained list of Data Object descriptors (whose max size is tentatively set at 8). The DOM Initialization routine is responsible for setting up the descriptor fields which essentially maps a given Data Object to a reserved set of Data Blocks (e.g., Flash Sectors or Pages) within Physical NVM in coordination with macro parameters defined in the NVM PAL.

Each Data Object descriptor consists of the following fields:

- **Descriptor String** – used to uniquely identify a given DO and ensuring that the meta-data/header in each Data Block is set up correctly.
- **Data Object Size** (in bytes) – Obviously, the number of Data Objects that can be stored in each Data Block is dependent on its byte size. Alternatively, the Data Object can also span multiple sectors (see next section).
- **Number of Allocated Blocks** – Determined by the initialization descriptor and NVM PAL macros for data block size. The DOM will round up to the nearest Data Block boundary.
- **Data Block Start/offset** – Determined by the order/total number of initialization descriptor(s) and the NVM PAL macros for defining which area of Physical NVM is reserved for use by the DOM. The start/offset is used as the initial start point for searching for the current/active Data Object in NVM.
- **Number of Data Objects per Data Block** – Determined by the DO Size and the NVM PAL macro that defines the size of the individual Data Blocks in NVM (i.e., sector/page size). This is essentially the DO capacity in NVM before the block needs to be erased/re-initialized – obviously, this is the critical factor in determining NVM wear-leveling over the lifespan of the device.
- **Data Block DO Offset** – The Data Block header (see next section) is variable length and is based on the number of DO that can fit into the Data Block. As a result, following initialization of the NVM resources, a quick offset pointer to the start of the DO record copies can be included to aid in determine the start of a valid/active DO.
- **Active DB** – Once the active a Data Object Fetch API is called, this variable is used as a quick access variable on subsequent Fetch and/or Commit calls from the meter application.
- **Active DO Slot** – Similar to the Active DB tracking variable with the exception that it references the Data Object offset in the current active Data Block.
- **Data Block Span Count** – This field is for large Binary Data Objects that span multiple NVM Blocks/Sectors (see next section). For smaller Data Objects that are less than the size of a sector, this value will always be one. NOTE: When this value is greater than one, the DOM will skip the number of sectors when scanning for valid DOM meta-data objects.

Once these descriptor(s) have been properly set up (and its associated meta-data headers in each Data Block NVM also verified/configured – see section on Data Block Management), subsequent API calls to the Data Object Manager (i.e., Fetch, Commit, Flush) will all used this information for determining which sector/offset in physical NVM to access the correct “active” Data Object for the application to act on.

## 3.3 Data Object Fragment Support

Alternatively, user-defined Data Objects managed by DOM can be quite large and will need to span multiple contiguous Blocks/Sectors in Physical NVM. The Data Object Manager handles this seamlessly as the following diagram shows:

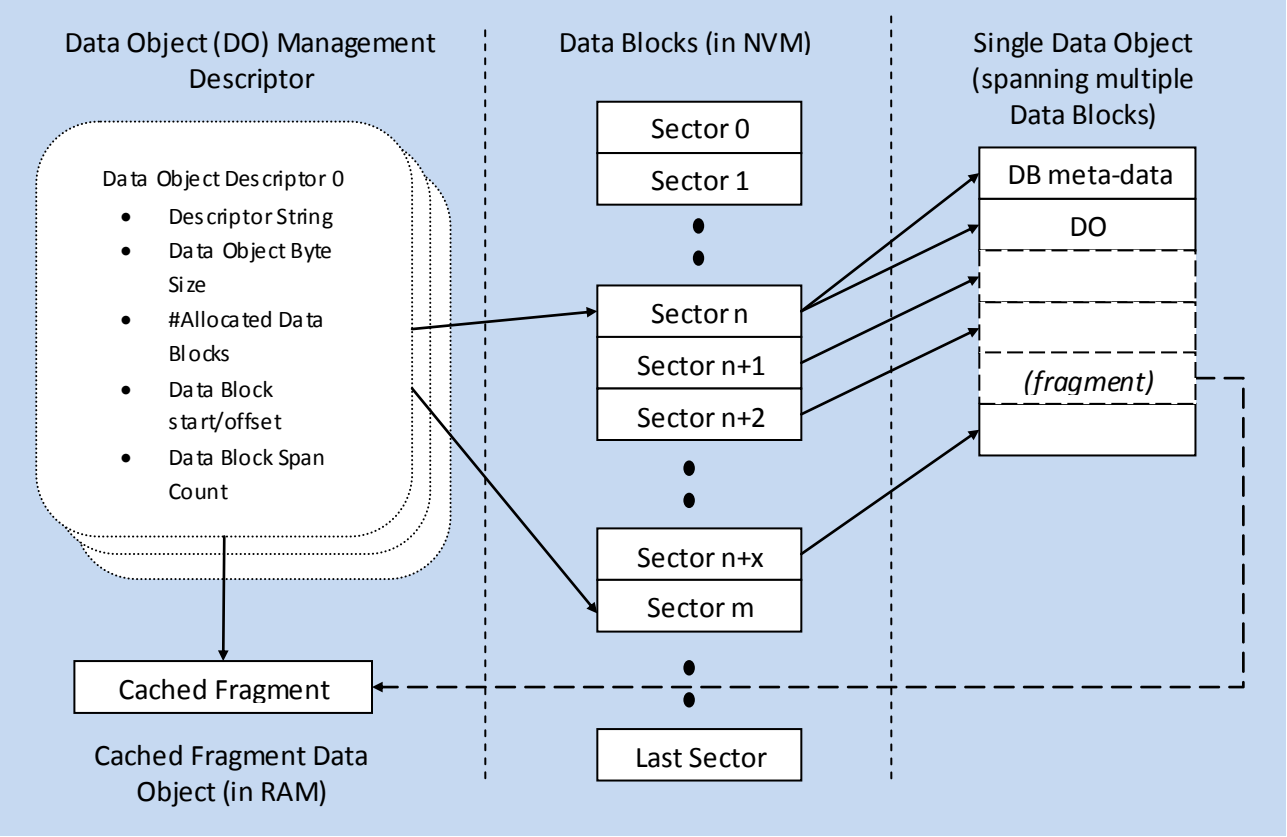


**Figure 3. Data Object Management (Data Object size > Data Block)**

As the diagram shows, the Data Block Span Count will be greater than one and sized based on the specified Data Object Size and the NVM sector size as reported by the NVM PAL software component.

Also, it may not be feasible for the application to keep an active cached copy of the entire Binary Data Object due to limited RAM memory – the Data Object Manager also supports a set of APIs for reading and writing specified "fragments" of the Data Object where the data size and offset is specified by the user (see API section for a list of said methods).

Regardless, of the size of the large data object, the Data Object Manager meta-data and management is the same.

## 3.4 Data Block Management

The organization of the Data Object copies in each Data Block within physical NVM consists of two parts:

- The Data Block header (i.e., meta-data) – This contains all of the information used by the DOM to identify and index the correct/current Data Object in NVM.
- The Data Object copies – essentially a chain of data objects from the end of the header to the end of the current block (i.e., sector and/or page size). Implementation details may necessitate that each record start at a word offset (e.g., word read/write accesses are more efficient then byte access).

One implementation detail regarding the header meta-data and the copy chain is that it assumes and takes advantage of the convention that all bit state changes for data stored in Flash are from '1'->'0' – otherwise a flash erase for the given Data Block will be required. It is also assumed when an NVM PAL erase method is called, that each bit in a given Data Block will revert to '1'. This convention is critical to the management of the meta-data which will need to be updated upon each Data Object Commit to NVM to correctly index the correct/active copy.

## 3.5 Data Block Header Fields

The following elements make up Data Block Header:

- **Data Object ID** – The index value of the DO descriptor the Data Block is associated with. If a mismatch is detected during DOM Initialization, the Data Block will be re-initialized. The expected size of this field is at least a 4-bit nibble. This field, once set, is *static* and will not change during subsequent Data Object Commits to NVM.
- **Data Block Full** – A Boolean indicating that all available Data Object slots have been used and the block will need to be re-initialized in order to be re-used. This a *dynamically* updated field – when set to digital '1', the block is *not* full. When set to '0', the block is full allowing the DOM Fetch and Commit APIs to quickly skip the current Data Block in its search for an active Data Object copy in Physical NVM.
- **Header Revision Marker** – The revision of the DO descriptor the Data Block is associated with. This allows the Data Object Manager to preserve data formats across evolving revisions of the Data Block Header (i.e., future-proofing - although this will be prevented as much as possible). Once set, this field is *static*.
- **Data Block Span Count** – The number of contiguous Data Blocks/Sectors that are managed by the header (i.e., for large Data Objects). Once set, this field is *static*.
- **Data Object String** – Limited to 8-characters and used as additional safety to ensure the Data Block is properly initialized and in sync with the Data Object Manager in the event that there are multiple data objects defined in the initialization descriptors that have identical byte lengths. Once set, this string identifier is *static*.
- **Data Object Size** – The size of the Data Object record, in bytes, saved in each copy "slot". If a mismatch is detected during DOM Initialization, the Data Block will be re-initialized (as it is considered invalid). The expected size of this field is 16-bits and, once set, is *static*.
- **Authentication** – A 4-byte field reserved for the authentication and/or encryption of the Data Object (currently unsupported).
- **Application Specific Data** – A two byte field reserved for use by the user application. The DOM provides APIs for both reading and setting this field. However, it is the user application responsibility to understand the NVM programming limitations when manipulating this variable (e.g., only '1'->'0' bit transitions on NOR Flash until the next erase cycle).
- **Data Object Offset Flags** – This is a dynamically sized set of Boolean flags – there is one offset flag for each Data Object copy that can fit in the current Data Block. Obviously the

byte length of this field is inversely proportional to the size of the object such that. This is a *dynamically* updated field – when an offset flag is set to digital '1', the Data Object copy is available for writing to. When set to '0', the Data Object copy has been written too. Also, the last '0' in the Data Object offset byte chain is the index to the last "active" Data Object in the current Data Block.

**NOTE:** The inverse relationship between Data Object Size and the number of bits to index the copy chain is perhaps the biggest drawback of this implementation – for example, a Data Block consisting of 1byte objects in a hypothetical Block size of 4K bytes would require a 512 byte overhead to reference each Data Object Copy. It is recommended that alternative resources on the MPU be used for small data objects such as this (e.g., File Register storage on the Kineitis MCU).

The following figure helps illustrate an example of the layout of a 4K-byte Data Block using the Data Block Structure described in this section for a 128-byte Data Object:

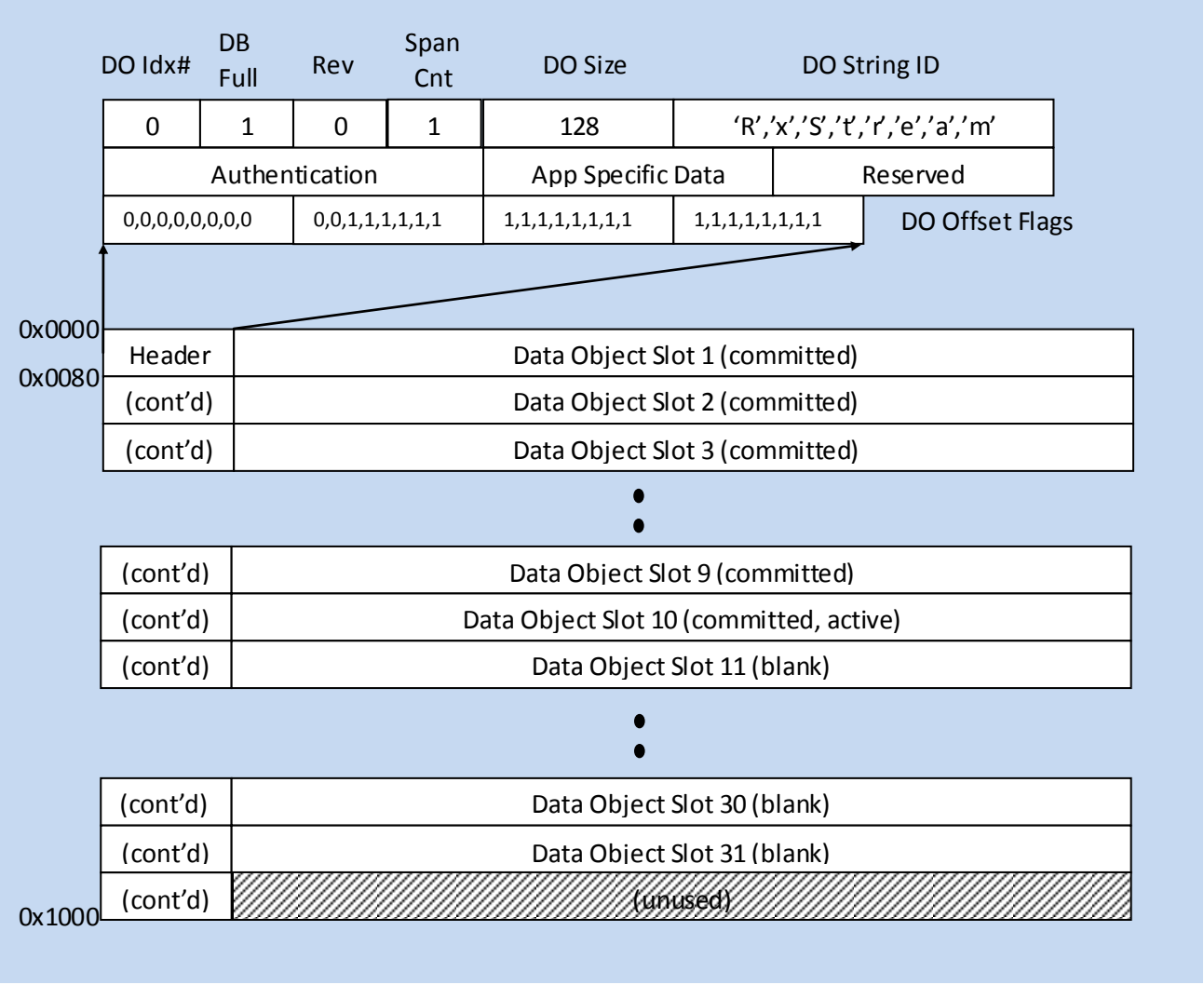


**Figure 4. Data Block Header Example**

In the above example, the header is 8-bytes and contains all of the DO object information that matches the Data object Management descriptor meta-data (see previous section). The location of the first Data Object is located at offset 0x0008 (and conveniently word-aligned – no alignment padding, if used, is necessary for this example).

The DB Full Flag is '1' indicating that the block is available for commits by the Data object Manager.

The DO Offset flags indicate that the 10th Data Object was the last Data Object slot committed too by the Data Object Manager – this is the active copy/slot and its contents will be copied into RAM following a Data Object Fetch call to the DOM software component.

If the Data Object Manager is invoked for a Data Object Commit, then the next Data Object slot (in this case, the 11th slot) will be written to using the NVM PAL Flash programming methods. Upon a successful update, then bit[10] in the 4-byte DO Offset Flag will be cleared indicating that it is now been updated and is now the "active" DO slot.

When all 31 Slots are committed and a subsequent Data Object Commit is issued, the DB Flag will be cleared (indicating that it is now full) and the Data Object Manager will look and the next consecutive Data Block for an available slot to write to.

## 3.6 Authentication Support

This design/feature is still in development.

# 4 DOM External Interfaces

## 4.1 Host Interface

The DOM implementation makes extensive use of host logging features to monitor the following:

- At the TRACE level, all API calls from the user application with their associated specified parameters (see next section for a comprehensive list of these APIs).
- At the INFO level, all Data Object initialization programming sequences.
- At the WARN level, any corrupted Data objects detected during the initialization process as well as user-level access levels (e.g., attempting to commit fragment data to a Data Object which has not been allocated).
- At the ERR level, all program sequence failures that results in lost/corrupted data – any condition that results in an ERR-level log will cause the DOM software component to revert to a non-initialized state.

In addition to logging, the DOM software unit provides the following monitor interfaces with the external Host PC (at least on the EMCM platform):

### 4.1.1 Data Object Config Message

Using this host↔node message configuration, an external user can verify the on-target DOM configuration as specified during the DOM initialization process in a human-readable format. On the EMCM platform, this is supported by using the following python script:

- emcm_get_data_obj_config.py

For each configured data object on the host application, this message will provide the following information:

- The Data Object string descriptor and Index.
- The Data Object size.
- The total allocated memory bytes for storage and wear-leveling of the data object. NOTE: This value takes into consideration both the meta-data overhead as well as the rounding up to the nearest sector boundary.
- The sector/block Index range in Physical NVM associated with the Data Object.

The following screenshot shows the response output from the host application for this message request as generated by the EMCM application:

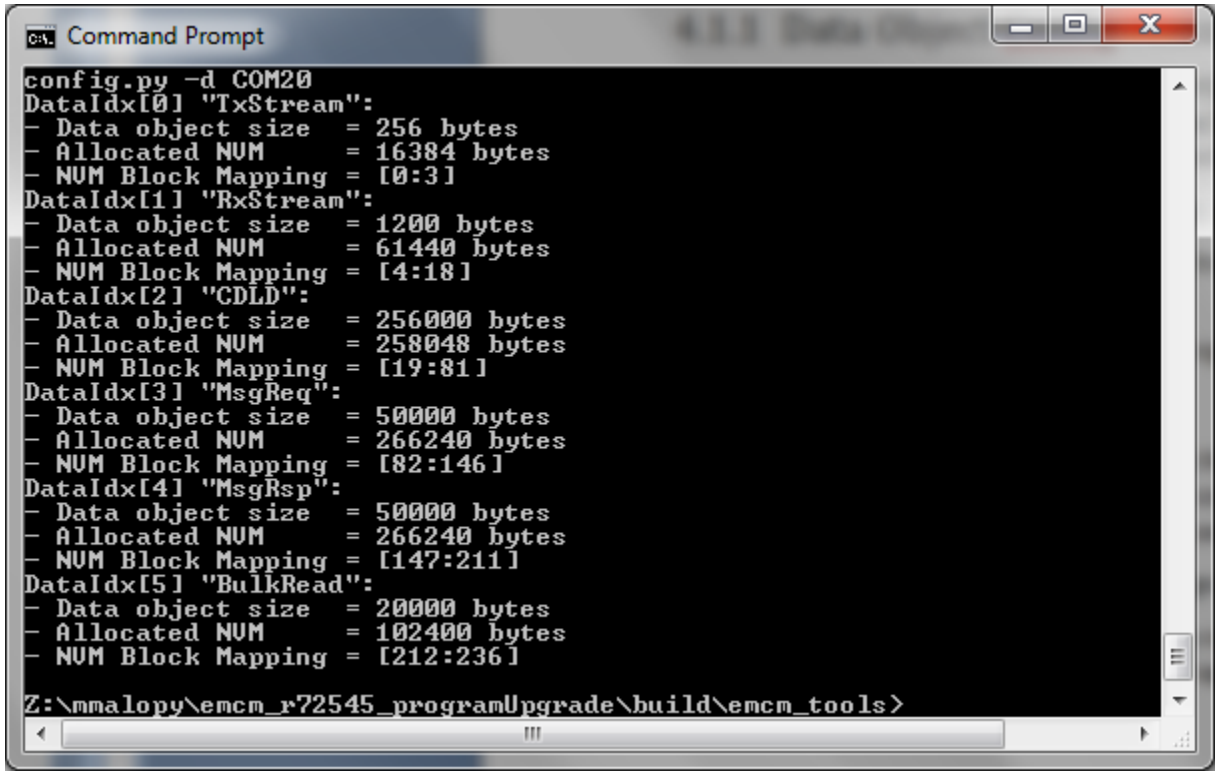


**Figure 5. Data Object Config Response Example**

NOTE: The screenshot shows the on-target configuration of the example shown in Appendix B.

## 4.1.2 Data Object Map Message

Using this host↔node message configuration, an external user can verify the DOM configured layout/organization of Physical NVM as specified during the DOM initialization process in a human-readable format. On the EMCM platform, this is supported by using the following python script:

- emcm_get_data_obj_map.py

For each managed Data Block in Physical NVM, the message will provide the following information as extracted from the Data Block's meta-data (see section 3.5).

- The Data Block Index (usually the physical sector ID).
- The Data Object string descriptor and Index from the Data Block's meta-data.
- The Data Blocks' FULL/NOT-FULL Boolean marker status.
- A visual representation of the status of each Data Object slot within the Data Block – i.e., empty, programmed, or error.

The following screenshot shows a partial response output from the host application for this message request as generated by the EMCM application (the response can be quite long):

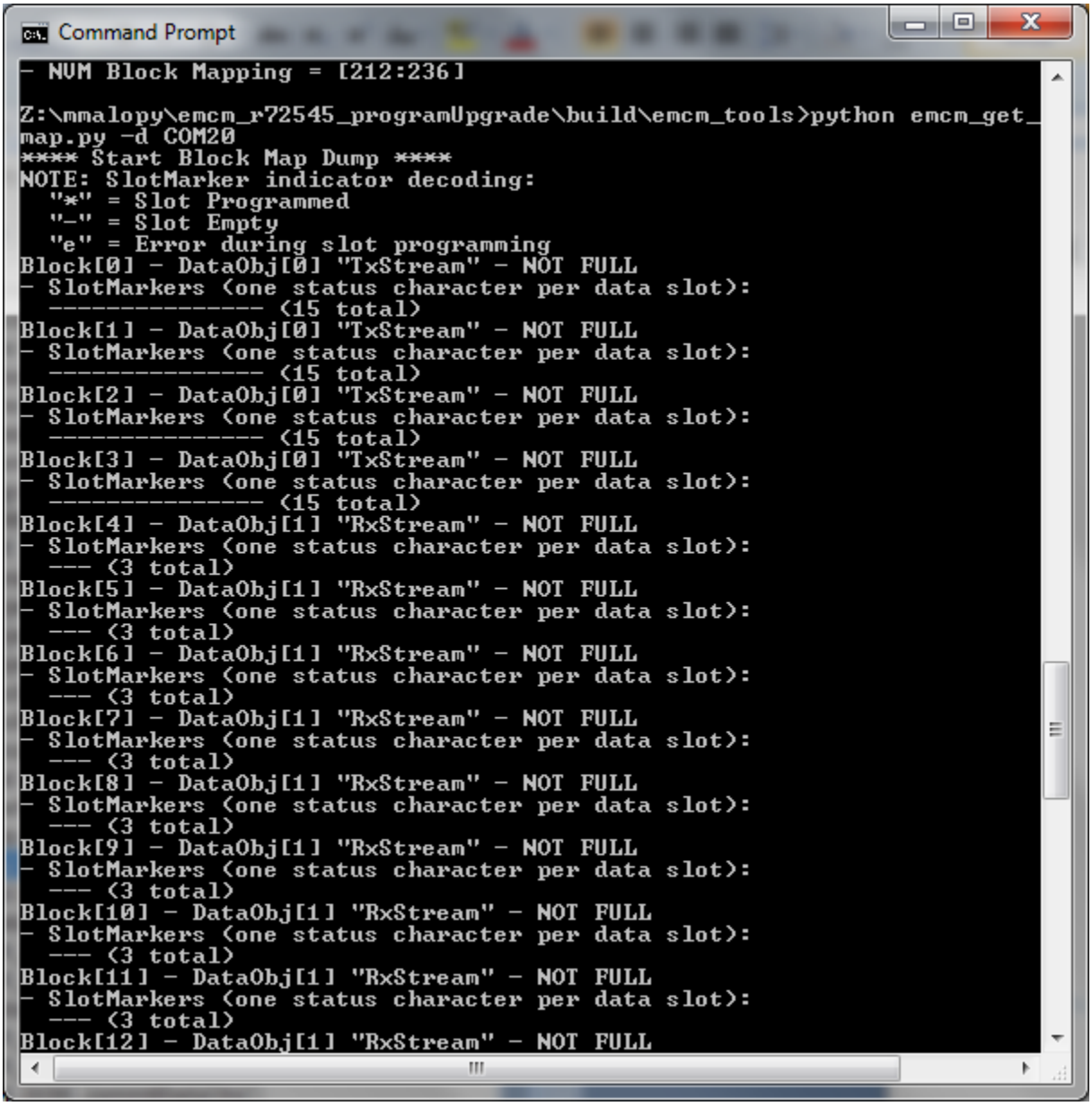


**Figure 6. Data Object Map Response Example**

NOTE: The screenshot shows the on-target configuration of the example shown in Appendix B.

# 5 DOM Internal Interfaces

## 5.1 Initialization/Setup APIs

The following APIs are used to support the initialization of the DOM software component.

### 5.1.1 NVM_DOM_init()

It is the responsibility of the User Application to call the DOM Initialization API when the host processor comes out of reset (e.g., Power-On Reset, Exception Reset, LLWU Reset, etc). All other DOM APIs will return an error until the initialization process has had a chance to configure the DOM meta-data for a particular Data Object.

This API will take the following parameter(s):

- A linked-list of descriptors data which are used to define how a Data Object is managed by the DOM which include:
  - String descriptor of the data object (for tracking purposes)
  - The size of the data object (in bytes).
  - The total size of physical NVM allocated for the data object wear leveling.
- Function callback to invoke when the re-entrant Init process has completed. NOTE: Setting this to NULL will cause the Init process to be execute in a blocking manner.
- Function callback to invoke if the NVM PAL handlers experience a write update failure (at which point the Init process will end and return a failure indication).
- An Init Bypass mask used to bypass the NVM Init process on selected Data Objects by DO Index. NOTE: Use with caution – misuse will result in corrupted Flash accesses.

This API will return the following:

- The Boolean status of the outcome of initialization – i.e., true when all of the data objects described by the descriptor(s) could be allocated - false if the descriptor(s) could not be processed.

The initialization is where most of the “heavy lifting” is performed within the Data Object Manager (DOM) by performing the following:

- Set up the Data Object Management meta-data that maps/allocates Data Objects to a set of Data Blocks (i.e., sectors and/or pages) in Physical NVM. Each Data Object will have an index number associated with (inferred by its element ordering in the data descriptors pass in as a parameter).
- Range check the Data Object Management meta-data such that the descriptors to not exceed the size allocated/defined in NVM PAL. A run-time assert will be generated under this condition.

- Check each Data Block for existing meta-data that is valid for a given Data Object – the intent of this check is to preserve previously updated copies of a Data Object committed to flash.
- Check each Data Block for an in-progress token indicating that an exception has occurred before the Data Object commit process could complete (e.g., an exception reset occurred on the target processor). If this exception state exists, then the last known active/good Data Object will be committed to the next active slot location in Physical NVM. If no previous active Data Object can be found, then the current/corrupted slot is marked as invalid.
- If a Data Block (i.e., sector or page) is blank or has invalid meta-data for a given Data Object it is assigned to, erase the Data Block and initialize its meta-data. In addition, warnings can be logged to track this type of operation.
- Once all Data Objects and its associated Data Blocks are confirmed and/or initialized, this API will return a Boolean true.

### 5.1.2 NVM_DOM_getNvmBlockOverhead()

This API aids the User Application in the creation of Data Objects that are precisely aligned with NVM Block/Sector boundaries by accounting for the DOM meta-data set up at the start of a Block/Sector in physical NVM.

This API has no parameters and be called regardless if the DOM software component is initialized or not (unlike other DOM APIs).

It will return the size of the overhead in external NVM, in bytes.

## 5.2 Core Data Management APIs

The following APIs are essential to properly managing and accessing any Data Object registered with the Data Object Manager.

### 5.2.1 NVM_DOM_checkDataObj()

This is provided for use by the User Application to determine if an active Data Object has been allocated in NVM for use by the application. It is used to support initial allocation/initialization verification.

This API will take the following parameter(s):

- A Data Object identifier – most likely the index of the Data Object in the descriptor list passed in at DOM initialization.

This API will return the following:

- The Boolean status indicating that the specified Data Object is allocated for use by the application.

### 5.2.2 NVM_DOM_getDataObj()

Used by the meter application to get the current/recent "active" copy of a given Data Object from Physical NVM. It is expected that this API will be called just following DOM Initialization so that the application can operate on a cached Data Object in RAM. Should a given Data Object *not* be found in Physical NVM, then it is the responsibility of the meter application to initialize a default Data Object in RAM to operate on.

This API will take the following parameter(s):

- A Data Object identifier – most likely the index of the Data Object in the descriptor list passed in at DOM initialization.
- A void pointer in RAM to update with the contents of the most recent Data Object from Physical NVM (if found).

This API will return the following:

- The Boolean status of the outcome of the data object fetch – i.e., true when the most recent given data object could be copied to the RAM pointer - false if the data object could not be found and/or processed.

The following processing steps describe the action(s) taken by this API:

- If the DOM software component has not been initialized, the API will immediately return a boolean false.
- With the aid of the Data Object Manager meta-data, the DOM software component will find the sector and byte offset of the most recent copy of a given Data Object from Physical NVM using the appropriate NVM PAL methods. If no copy can be found, a Boolean False is returned.
- If found, the DOM software component will copy the given Data Object from Physical NVM to its given cache area in RAM. NOTE: It is the responsibility of the meter application to ensure that the cache area is properly allocated (i.e., it does not overrun other RAM data objects).
- Only until the Data Object has been successfully transferred from Physical NVM to RAM will the API return with a boolean true.

### 5.2.3 NVM_DOM_commitDataObj()

This API is used by the meter application when state changes in a given cached Data Object in RAM (including default initialization) is required to be updated in persistent NVM.

This API will take the following parameter(s):

- A Data Object identifier – most likely the index of the Data Object in the descriptor list passed in at DOM initialization.
- A void pointer in RAM where the cached copy of the Data Object is located.

This API will return the following:

- The Boolean status of the outcome of the data object commit – i.e., true when the data object and associated meta-data are updated in physical NVM - false if the commit operation could not be processed.

The following processing steps describe the action(s) taken by this API:

- If the DOM software component has not been initialized, the API will immediately return a boolean false.
- With the aid of the Data Object Manager meta-data, the DOM software component will find the sector and byte offset of the next available space in Physical NVM to update with the cached RAM copy. If necessary, the DOM software component will clear/erase a data block in NVM to make room for the given Data Object. If the update in Physical NVM cannot be allocated or failed, a Boolean False is returned.
- Once the sector and byte offset is determined, the DOM software component will program the given Data Object from its cache area in RAM into its offset in Physical NVM using the appropriate NVM PAL methods. If the program update fails, a Boolean False is returned.
- Once the Data object has been committed to Physical NVM, the Data Block meta-data, also located in Physical NVM (see Data Block Management section), will also need to be updated to reflect the location of the new "active" Data Object offset in NVM using the appropriate NVM PAL Methods. If the program update fails, a Boolean False is returned. NOTE: Because a commit is a two-stage update in Flash, any exception condition in the programming sequence (e.g., a system reset) will need to be detected and corrected through manipulation of the DOM meta-data – A recovery mechanism during the Init process will revert DOM to the last known good "active" block then to have a possible corrupted "active" block in NVM.
- Only until both the active Data Object and associated meta-data are updated in memory will the API return with a Boolean true.

### 5.2.4 NVM_DOM_overwriteDataObj()

This API is similar to NVM_DOM_commitDataObj() (see previous API) in that it updates NVM with the contents of a given Data Object in RAM. However, unlike the commit method, it overwrites the current active NVM slot – this is most useful for "NVM-Aware" manipulation of Data Objects that follow the conventions/limitation of physical NVM.

This API will take the following parameter(s):

- A Data Object identifier – most likely the index of the Data Object in the descriptor list passed in at DOM initialization.
- A void pointer in RAM where the cached copy of the Data Object is located.

This API will return the following:

- The Boolean status of the outcome of the data object overwrite – i.e., true when the data object and associated meta-data are updated in physical NVM - false if the overwrite operation could not be processed.

The following processing steps describe the action(s) taken by this API:

- If the DOM software component has not been initialized, the API will immediately return a boolean false.
- With the aid of the Data Object Manager meta-data, the DOM software component will find the sector and byte offset of the current active slot in Physical NVM to update with the cached RAM copy. If no active slot is found in Physical NVM, a Boolean False is returned.
- Once the sector and byte offset is determined, the DOM software component will program the given Data Object from its cache area in RAM into its offset in Physical NVM using the appropriate NVM PAL methods. If the program update fails, a Boolean False is returned.
- Once the active Data Object has been overwritten in memory will the API return with a Boolean true.

### 5.2.5 NVM_DOM_flushDataObj()

This API is used to clear/erase all of Physical NVM associated with a given Data Object and re-initialize the associated Node Block meta-data. This can be used to support transitions between image and/or meter upgrades where the meta-data format has changed.

This API will take the following parameter(s):

- A Data Object identifier – most likely the index of the Data Object in the descriptor list passed in at DOM initialization.

This API will return the following:

- The Boolean status of the outcome of the data object flush – i.e., true when the data object and associated meta-data are re-initialized in physical NVM - false if the flush operation could not be processed.

The following processing steps describe the action(s) taken by this API:

- If the DOM software component has not been initialized, the API will immediately return a boolean false.
- Using the DOM Data Object Manager meta-data, all Data Blocks mapped to the given Data Object will be erased using the appropriate NVM PAL methods. If the erase update(s) fail, a Boolean False is returned.
- Once all associated Data Blocks in Physical NVM are erased, the Data Block meta-data in each sector will be re-initialized for the given Data Object – again, using the appropriate NVM PAL methods. Should any of the program updates to NVM fail, then a Boolean false is returned.
- Only until all Data Blocks are erased and the associated meta-data are updated in memory will the API return with a Boolean true.

### 5.2.6 NVM_DOM_getDataObjBase()

In the majority of the interactions between the User Application and the DOM software component, the actual physical location/address of the active Data Object is not exposed. However, this API is provided if direct access to physical NVM is required outside of the normal API methods (e.g., EMCM Code Download cutover support that requires direct access by functions executing out of RAM).

This API will take the following parameter(s):

- A Data Object identifier – most likely the index of the Data Object in the descriptor list passed in at DOM initialization.
- A pointer to a NVM Block/Sector index variable to update for the active/specified Data Object. It is the responsibility of the user to specify a proper memory location for this 16-bit index variable.
- A pointer to a byte offset variable to update for the active/specified Data object. Again, it is the responsibility of the user to specify a proper memory location for this 32-bit offset variable.

This API will return the following:

- The Boolean status indicating that the specified Data Object has been found and the corresponding index/offset variables have been updated.

## 5.3 Fragment Management APIs

The following APIs are essential to properly managing and accessing any Large Binary Data Object registered with the Data Object Manager – a Large Binary Data Object is defined as user data that cannot be accessed in its entirety by the User Application due to RAM limitations and will span multiple NVM blocks/sectors in physical NVM.

### 5.3.1 NVM_DOM_allocateDataObj()

This API is used by the meter application to allocate/reserve a given Data Object slot in persistent NVM without programming any portion of the data. It is primarily intended for the management of larger binary data objects where a cached copy of the Data Object in RAM is not viable. Follow up management of the data in the allocated/reserved slot would be done via fragment commits and reads (see subsequent API sections). Similar to the DOM initialization process, this method supports both a blocking and non-blocking API mode due to the possibility of a lengthy bulk erase process to prepare the next Data Object slot.

This API will take the following parameter(s):

- A Data Object identifier – most likely the index of the Data Object in the descriptor list passed in at DOM initialization.
- An optional function callback to invoke when the re-entrant Allocate process has completed. NOTE: Setting this to NULL will cause the Allocate process to be execute in a blocking manner.

This API will return the following:

- The Boolean status of the outcome of the data object allocation – i.e., true when the associated meta-data is updated in physical NVM (update pending if using the non-blocking method)- false if the allocation operation could not be processed.

The following processing steps describe the action(s) taken by this API:

- If the DOM software component has not been initialized, the API will immediately return a boolean false.
- With the aid of the Data Object Manager meta-data, the DOM software component will find the sector and byte offset of the next available space in Physical NVM to reserve as the active DO slot. If necessary, the DOM software component will clear/erase the data block(s) in NVM to make room for the given Data Object. If the update in Physical NVM cannot be allocated or failed, a Boolean False is returned. If the callback field is non-NULL, then this check and/or erase is done using the re-entrant capabilities of the NVM PAL driver.
- Only until the Data Object slot is reserved and the associated meta-data are updated in memory will the API return with a Boolean true or the registered re-entrant callback be invoked.

### 5.3.2 NVM_DOM_getDataObjFragment()

Used by the meter application to get a specified data fragment for the current/recent "active" copy of a given Data Object from Physical NVM.

This API will take the following parameter(s):

- A Data Object identifier – most likely the index of the Data Object in the descriptor list passed in at DOM initialization.
- A byte offset into the Data Object to begin the data fragment copy from Physical NVM into RAM.
- The byte length of the date fragment copy from Physical NVM into RAM.
- A void pointer in RAM to update with the data fragment from the most recent Data Object from Physical NVM (if found).

This API will return the following:

- The Boolean status of the outcome of the data object fragment copy – i.e., true when the specified fragment of a given data object could be copied to the RAM pointer - false if the specified fragment of a given data object could not be found and/or processed.

The following processing steps describe the action(s) taken by this API:

- If the DOM software component has not been initialized, the API will immediately return a boolean false.
- With the aid of the Data Object Manager meta-data, the DOM software component will find the sector and byte offset of the most recent copy of a given Data Object from

Physical NVM using the appropriate NVM PAL methods. If no copy can be found, a Boolean False is returned.

- If found, the DOM software component will copy the specified data fragment of the given Data Object from Physical NVM to its given cache area in RAM. NOTE: It is the responsibility of the user application to ensure that the cache area is properly allocated (i.e., it does not overrun other RAM data objects).
- Only until the Data Object fragment has been successfully transferred from Physical NVM to RAM will the API return with a boolean true.

### 5.3.3 NVM_DOM_writeDataObjFragment()

Used by the meter application to commit/program a specified data fragment to the current/recent "active" copy of a given Data Object in Physical NVM.

This API will take the following parameter(s):

- A Data Object identifier – most likely the index of the Data Object in the descriptor list passed in at DOM initialization.
- A void pointer in RAM containing the data fragment to copy/program into the most recent Data Object in Physical NVM (if found).
- A byte offset into the Data Object to begin the data fragment copy/programming sequence from RAM into Physical NVM.
- The byte length of the date fragment copy from RAM into Physical NVM.

This API will return the following:

- The Boolean status of the outcome of the data object write – i.e., true when the data fragment is updated in physical NVM - false if the write operation could not be processed.

The following processing steps describe the action(s) taken by this API:

- If the DOM software component has not been initialized, the API will immediately return a boolean false.
- With the aid of the Data Object Manager meta-data, the DOM software component will find the sector and byte offset of the most recent copy of a given Data Object from Physical NVM using the appropriate NVM PAL methods. If no copy can be found, a Boolean False is returned.
- Once the sector and byte offset is determined, the DOM software component will program the given data fragment from its cache area in RAM into its offset in Physical NVM using the appropriate NVM PAL methods. If the program update fails, a Boolean False is returned.
- If the program update fails or the fragment offset and/or length goes beyond the bounds of the specified Data Object, a Boolean False is returned.
- Only until the Data Object fragment has been successfully programmed into Physical NVM will the API return with a boolean true.

# 5.4 Application Specific Meta-Data APIs

The following APIs are provided for use by the User Application to manipulate a persistent reserved field in a Data Block's meta-data for use in managing a given Data Object. An example use of application specific meta-data would be the size of a variable length object programmed into the NVM active slot (providing it does not exceed the max size of the data object). All manipulation of the application-specific meta-data will need to conform to program limitations of the Physical NVM in which the meta-data is located (e.g., '1'->'0' bit transitions in NOR flash).

## 5.4.1 NVM_DOM_getAppSpecificDataSize()

This is provided for use by the application to determine the size of application-specific meta-data for it to manipulate use.

There are no parameters associated with this API. However, it will return the following:

- The size of the application-specific meta-data field for all Data Objects, in bytes.

## 5.4.2 NVM_DOM_getAppSpecificMetaData()

This API is provided to the user application to get application-specific meta-data from the current/recent "active" Data Block meta-data header from Physical NVM (see section 3.5).

This API will take the following parameter(s):

- A Data Object identifier – most likely the index of the Data Object in the descriptor list passed in at DOM initialization.
- A void pointer in RAM to update with the contents of the application-specific meta-data field from Physical NVM (if found).

This API will return the following:

- The Boolean status of the outcome of the meta-data fetch – i.e., true when meta-data could be copied to the RAM pointer - false if the data block header could not be found and/or processed.

The following processing steps describe the action(s) taken by this API:

- If the DOM software component has not been initialized, the API will immediately return a boolean false.
- With the aid of the Data Object Manager meta-data, the DOM software component will find the sector and byte offset of the most recent copy of a given Data Block meta-data header from Physical NVM using the appropriate NVM PAL methods. If no copy can be found, a Boolean False is returned.
- If found, the DOM software component will copy only the application-specific data from Physical NVM to its given cache area in RAM. NOTE: It is the responsibility of the meter application to ensure that the cache area is properly allocated (i.e., it does not overrun other RAM data objects).

- Only until the meta-data has been successfully transferred from Physical NVM to RAM will the API return with a boolean true.

### 5.4.3 NVM_DOM_setAppSpecificMetaData()

This API is provided to the user application to set application-specific meta-data to the current/recent "active" Data Block meta-data header in Physical NVM (see section 3.5).

This API will take the following parameter(s):

- A Data Object identifier – most likely the index of the Data Object in the descriptor list passed in at DOM initialization.
- The actual application-specific meta-data to be programmed into Physical NVM.

This API will return the following:

- The Boolean status of the outcome of the meta-data commit – i.e., true when meta-data could be programmed into Physical NVM - false if the data block header could not be found and/or programmed.

The following processing steps describe the action(s) taken by this API:

- If the DOM software component has not been initialized, the API will immediately return a boolean false.
- With the aid of the Data Object Manager meta-data, the DOM software component will find the sector and byte offset of the most recent copy of a given Data Block meta-data header from Physical NVM using the appropriate NVM PAL methods. If no copy can be found, a Boolean False is returned.
- If found, the DOM software component will program the application-specific data into the appropriate Data Block header offset in Physical NVM. NOTE: This update to the Data Block Header will not alter or change other meta-data fields nor will it alter any user Data Object slots.
- Only until the application-specifiec meta-data has been successfully programmed into Physical NVM will the API return with a boolean true.

# Appendix A Abbreviations and Terms

| Abbreviation/Term | Definition |
|---|---|
| AP | Access Point. The ULP network component geographically deployed over a territory. |
| DB | Data Block |
| DO | Data Object |
| DOM | Data Object Manager |
| EMS | Element Management System. The network component that provides a concise view of the ULP network for controls and alarms. |
| FMC | Flash Memory Controller. A Kinetis module responsible for managing Program Flash, Data Flash, and/or FlexNVM. |
| GW | Gateway. The network appliance that provides a single entry point into the back office for the ULP network. A gateway talks upstream to the EMS and CIMA. It talks downstream to multiple APs. |
| HAL | Hardware Abstraction Layer. A software functional block on the ACM used to implement drivers for all ULP node interfaces. |
| HLD | High Level Design |
| IRQ | Interrupt Request |
| MCU | Microcontroller Unit |
| Node | The generic term used interchangeably with an end point On-Ramp Wireless device. |
| NVM | Non-Volatile Memory. In the context of the Kinetis, memory that is persistent across the different Low Leakage power modes. |
| ORW | On-Ramp Wireless |
| OTA | Over-the-Air |
| RAM | Random Access Memory |
| RTOS | Real Time Operating System |
| ULP | Ultra-Link Processing™. The On-Ramp Wireless' proprietary wireless communication technology. |
| UNIL | ULP Node Interface Library. The On-Ramp Wireless software library containing all of the necessary software interfaces to communicate with the On-Ramp Wireless Node. |
| uNode | Shorthand for microNode. The microNode is a second generation, small form factor, ULP wireless network module developed by On-Ramp Wireless that works in combination with various devices and sensors and communicates data to an Access Point. |
| UTC | Universal Time Coordinated. The time standard maintained on the On-Ramp wireless network. |

# Appendix B Example DOM Init Configuration

This appendix can be referenced to illustrate an example use of the Data Object Manager to support the EMCM application – In this case, the EMCM requires an external 8MB SPI-FLASH to store the following persistent data objects:

- ULP Stream Protocol (USP) Tx Stream State – 50 x 256byte slots in NVM to provide the necessary wear leveling of Tx Stream State over the lifetime of the EMCM product.
- ULP Stream Protocol (USP) Rx Stream State – 50 x 1.2Kbyte slots in NVM to provide the necessary wear leveling of Rx Stream State over the lifetime of the EMCM product.
- Code Download (CDLD) buffer – 1 x 256kByte slot in NVM utilizing the fragment DOM API methods.
- Downlink C12.19 Message Request Buffer – 5 x 50k slots in NVM to provide the necessary wear leveling of C12.19 Requests over the lifetime of the EMCM product.
- Uplink C12.19 Message Response Buffer – 5 x 50k slots in NVM to provide the necessary wear leveling of C12.19 Responses over the lifetime of the EMCM product.
- Meter Bulk Read Scratch Buffer – 5x20k slots in NVM to provide the necessary wear leveling of the bulk read storage area (used in support of a C12.19 Program Sequence).

The DOM Initialization descriptor list for the above data objects when be set up as so:

```
initDescriptor[0].descriptorString = “TxStream”;
initDescriptor[0].dataObjSize = 256;
initDescriptor[0].allocNvmMem = (50 * 256);

initDescriptor[1].descriptorString = “RxStream”;
initDescriptor[1].dataObjSize = 1200;
initDescriptor[1].allocNvmMem = (50 * 1200);

initDescriptor[2].descriptorString = “CDLD”;
initDescriptor[2].dataObjSize = 256000;
initDescriptor[2].allocNvmMem = 256000;

initDescriptor[3].descriptorString = “MsgReq”;
initDescriptor[3].dataObjSize = 50000;
initDescriptor[3].allocNvmMem = (5 * 50000);

initDescriptor[4].descriptorString = “MsgRsp”;
initDescriptor[4].dataObjSize = 50000;
initDescriptor[4].allocNvmMem = (5 * 50000);

initDescriptor[5].descriptorString = “BulkRead”;
initDescriptor[5].dataObjSize = 20000;
initDescriptor[5].allocNvmMem = (5 * 20000);
```

After initializing an initDescriptor[6] element to all “0”s, all that is needed to configure external NVM for the above initialization descriptor is the following call:

```
initStatus = NVM_DOM_init(initDescriptor, NULL, NULL, 0); /* see section 5.1.1 for the API description */
```