www.qorvo.com
lpw.support@qorvo.com

gpNvm Reference Manual

API Description

ElemIf version

Version 2.10.2.0
November 15, 2021

Contents

Chapter 1

Introduction

This document describes in a formal manner the API interface that can be used to control all the functionality of the Non-Volatile Memory (NVM) component.

The document covers the element based interface.

Chapter 2

Module Documentation

2.1 General NVM functions

The general NVM functionality is implemented in these functions.

Functions

- void gpNvm_RegisterSection (UInt8 componentId, const ROM gpNvm_Tag_t *tags, UInt8 nbrOfTags, gpNvm_cbCheckConsistency_t cbCheckConsistency)

  Registers a component to the NVM software block.

2.1.1 Detailed Description

Checking and clearing the NVM is made possible by these functions. The implementation of these functions can be unique for each type of NVM used.

2.1.2 Function Documentation

gpNvm_RegisterSection()

```
void gpNvm_RegisterSection (
            UInt8 componentId,
            const ROM gpNvm_Tag_t * tags,
            UInt8 nbrOfTags,
            gpNvm_cbCheckConsistency_t cbCheckConsistency )
```

Registers a component to the NVM software block. This will define a NVM section. The function should be called from the _init function of the component.

Parameters

componentId	An unique idenditfier of the registered section, which will be used further on to refer to this section. The GP_COMPONENT_ID should be used for this.
gpNvm_Tag_t	A pointer to a gpNvmTag_t array allocated in flash contining the information of all nvm-tags of the registered section.
nbrOfTags	The number of tags in the gpNvm_Tag_t array tags.

Parameters

cbCheckConsistency	A function pointer to a callback which will be called by the nvm-component if it's performing a consistency check. This allows specific checks on the content of the section which are only known by the component. If specified as NULL, this additional check will be ommitted.

2.2 Element interface NVM functions

This file contains element specific interface to the gpNvm component.

Functions

- void gpNvm_Config (const ROM gpNvm_LookUpTableHeader_t ∗baseTableHeader, const ROM gpNvm_ElementDefine_t ∗baseTableElements, const ROM gpNvm_VersionCrc_t ∗inheritedNvmVersion

    Configures NVM for base tables other than gpNvm_elementBaseLUTHeader and gpNvm_elementBaseLUTElements.
- void gpNvm_RegisterElement (const ROM gpNvm_IdentifiableTag_t ∗pTag)

    Registers an element to the NVM software block.
- void gpNvm_RegisterElements (const ROM gpNvm_IdentifiableTag_t ∗pTag, UInt8 nbrOfTags)

    Registers multiple elements to the NVM software block.

2.2.1 Detailed Description

This header is implicitly included via the main interface header if the element interface diversity is selected. The element interface version of the gpNvm component treats various NVM attributes in a different way than the standard gpNvm version. In the standard version the components register arrays of tags for the NVM attributes which form sections belonging to its component. Tag IDs are derived implicitly from the index of a tag within the section array and if application was to configure a component in such way that attributes are added or dropped, the implicit tag IDs could change. In addition to that, the attributes are placed in the NVM one after another without embedding information to help disambiguate them when the firmware changes.

The element interface offers solution to this problem having in mind also the possibility to reuse inherited standard NVM by the element interface compatible application. The following image illustrates the reference situation (NVM usage of last non-up/downgradeable firmware) and relationship of it to the unique IDs of the attributes:

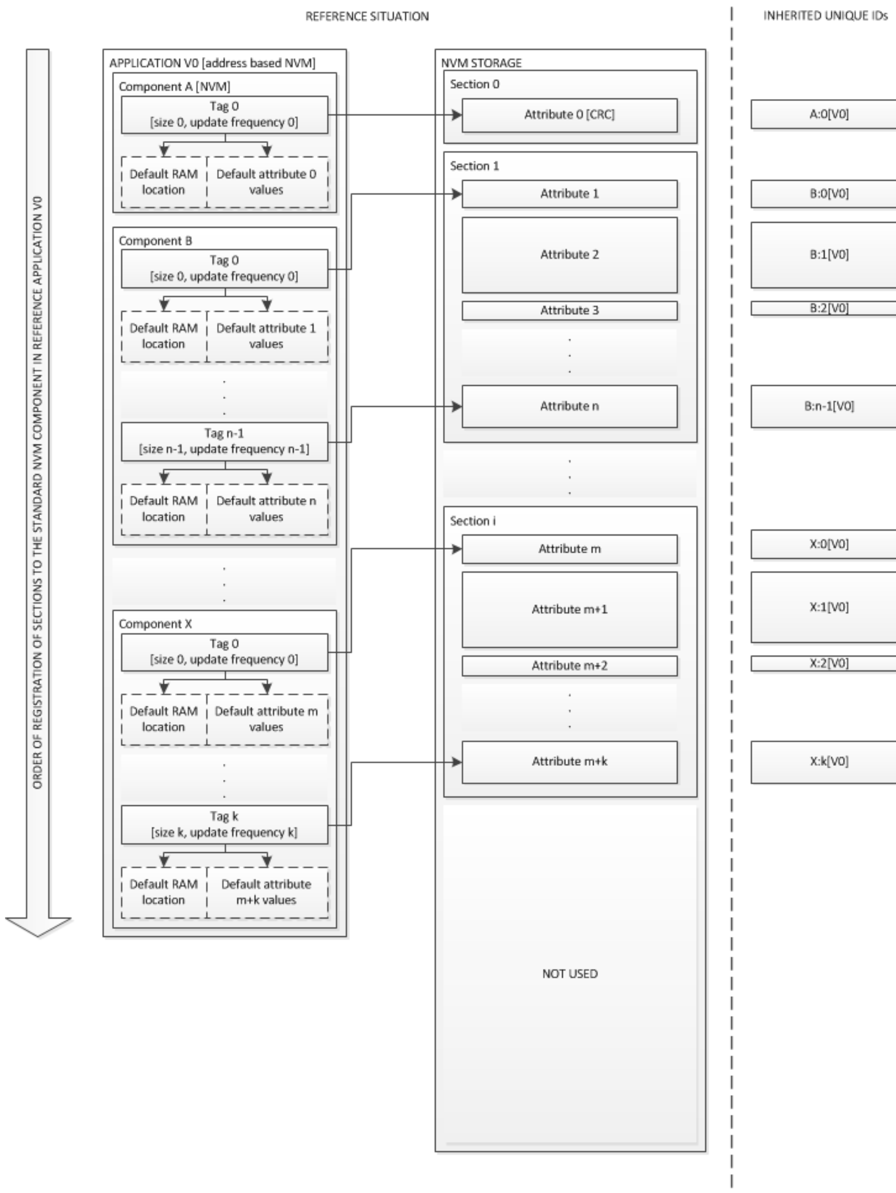

REFERENCE SITUATION
INHERITED UNIQUE IDs
ORDER OF REGISTRATION OF SECTIONS TO THE STANDARD NVM COMPONENT IN REFERENCE APPLICATION V0
APPLICATION V0 [address based NVM]
Component A [NVM]
Tag 0 [size 0, update frequency 0]
Default RAM location
Default attribute 0 values
Component B
Tag 0 [size 0, update frequency 0]
Default attribute 1 values
Tag n-1 [size n-1, update frequency n-1]
Default attribute n values
Component X
Default attribute m values
Tag k [size k, update frequency k]
Default attribute m+k values
NVM STORAGE
Section 0
Attribute 0 [CRC]
Section 1
Attribute 1
Attribute 2
Attribute 3
Attribute n
Section i
Attribute m
Attribute m+1
Attribute m+2
Attribute m+k
NOT USED
A:0[V0]
B:0[V0]
B:1[V0]
B:2[V0]
B:n-1[V0]
X:0[V0]
X:1[V0]
X:2[V0]
X:k[V0]

Figure 2.1: Reference NVM and mapping to UIDs

To resolve the issue of lacking the information about the attributes in the inherited standard NVM, the application is required to provide this information in a form of a program memory look-up table as illustrated in the following image:

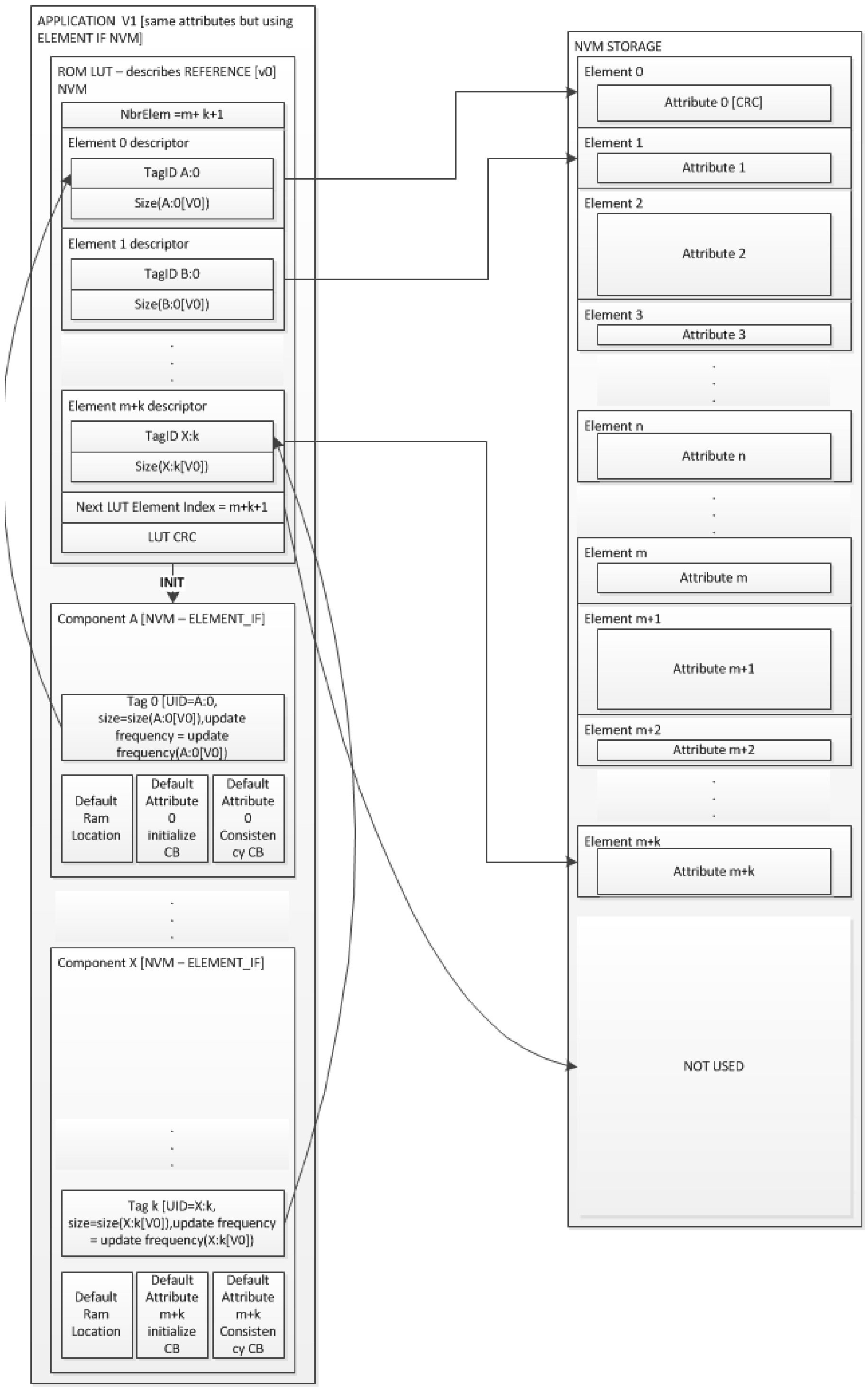

APPLICATION V1 [same attributes but using ELEMENT IF NVM]
ROM LUT – describes REFERENCE [v0] NVM
NbrElem =m+ k+1
Element 0 descriptor
TagID A:0
Size(A:0[V0])
Element 1 descriptor
TagID B:0
Size(B:0[V0])
Element m+k descriptor
TagID X:k
Size(X:k[V0])
Next LUT Element Index = m+k+1
LUT CRC
INIT
Component A [NVM – ELEMENT_IF]
Tag 0 [UID=A:0, size=size(A:0[V0]),update frequency = update frequency(A:0[V0])]
Default Ram Location
Default Attribute 0 initialize CB
Default Attribute 0 Consistency CB
Component X [NVM – ELEMENT_IF]
Tag k [UID=X:k, size=size(X:k[V0]),update frequency = update frequency(X:k[V0])]
Default Ram Location
Default Attribute m+k initialize CB
Default Attribute m+k Consistency CB
NVM STORAGE
Element 0
Attribute 0 [CRC]
Element 1
Attribute 1
Element 2
Attribute 2
Element 3
Attribute 3
Element n
Attribute n
Element m
Attribute m
Element m+1
Attribute m+1
Element m+2
Attribute m+2
Element m+k
Attribute m+k
NOT USED

Figure 2.2: Element interface based application and inherited NVM

Element interface based application is allowed to add/remove attributes and change attribute sizes (with limitation that the common denominator always stays alligned to the beginning of the attribute). The information about the added content into the NVM is embedded in the NVM as linked list of look-up tables. Initial program memory based look-up table is used as starting point for parsing of this linked list. Each time some component registeres attribute not known already in the NVM, NVM look-up table gets extended with this information and new attribute gets its space reserved in the NVM. Look-up tables are NVM elements themselves and as such are limited in size so the link list chaining is provided as mechanism for further extension.

The following images illustrate several typical situations of firmware upgrades and downgrades causing redefinition of the attributes:

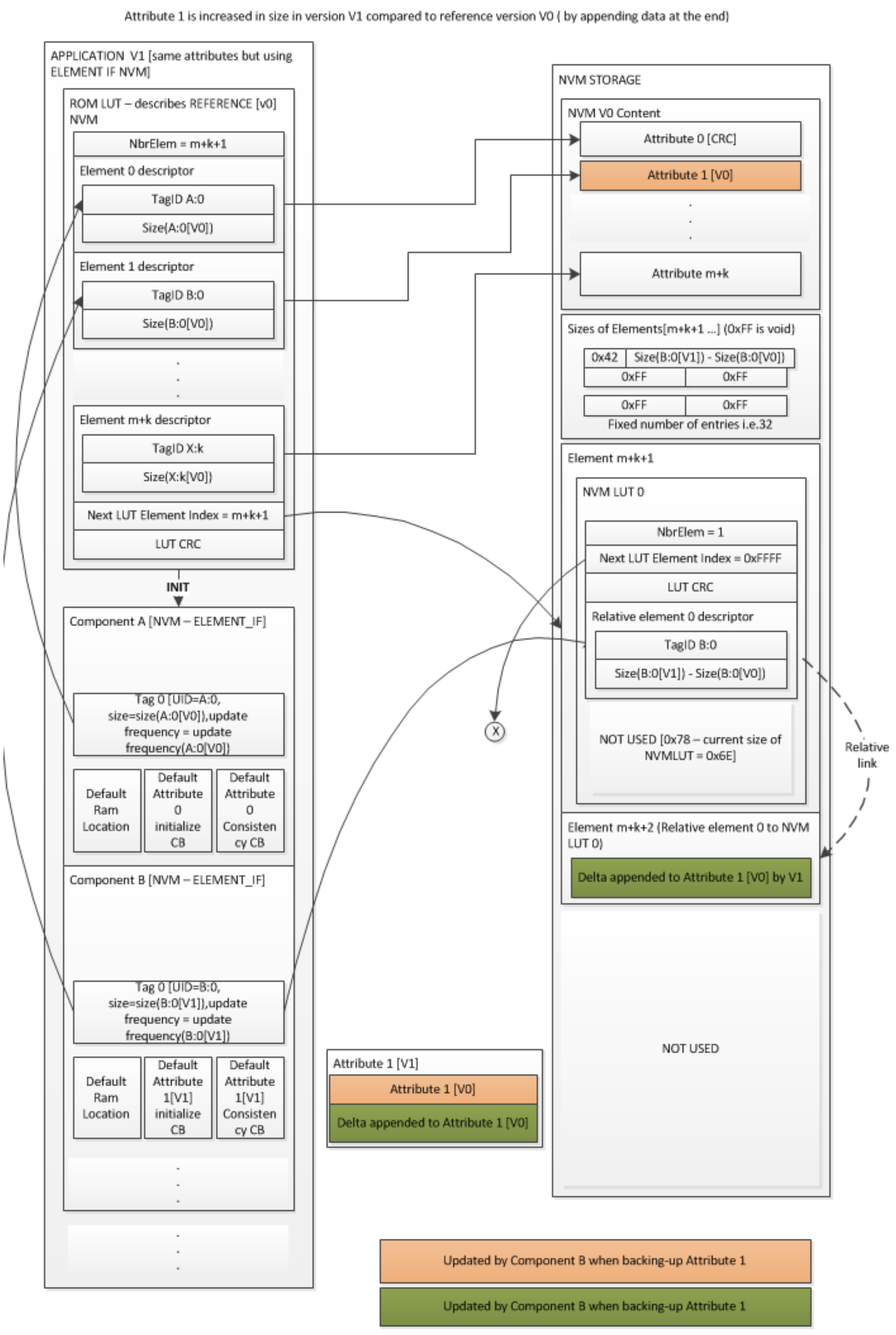

Attribute 1 is increased in size in version V1 compared to reference version V0 (by appending data at the end)
APPLICATION V1 [same attributes but using ELEMENT IF NVM]
ROM LUT – describes REFERENCE [v0] NVM
NbrElem = m+k+1
Element 0 descriptor
TagID A:0
Size(A:0[V0])
Element 1 descriptor
TagID B:0
Size(B:0[V0])
Element m+k descriptor
TagID X:k
Size(X:k[V0])
Next LUT Element Index = m+k+1
LUT CRC
INIT
Component A [NVM – ELEMENT_IF]
Tag 0 [UID=A:0, size=size(A:0[V0]),update frequency = update frequency(A:0[V0])]
Default Ram Location
Default Attribute 0 initialize CB
Default Attribute 0 Consistency CB
Component B [NVM – ELEMENT_IF]
Tag 0 [UID=B:0, size=size(B:0[V1]),update frequency = update frequency(B:0[V1])]
Default Ram Location
Default Attribute 1[V1] initialize CB
Default Attribute 1[V1] Consistency CB
NVM STORAGE
NVM V0 Content
Attribute 0 [CRC]
Attribute 1 [V0]
Attribute m+k
Sizes of Elements[m+k+1 ...] (0xFF is void)
0x42
Size(B:0[V1]) - Size(B:0[V0])
0xFF
0xFF
0xFF
0xFF
Fixed number of entries i.e.32
Element m+k+1
NVM LUT 0
NbrElem = 1
Next LUT Element Index = 0xFFFF
LUT CRC
Relative element 0 descriptor
TagID B:0
Size(B:0[V1]) - Size(B:0[V0])
NOT USED [0x78 – current size of NVMLUT = 0x6E]
X
Relative link
Element m+k+2 (Relative element 0 to NVM LUT 0)
Delta appended to Attribute 1 [V0] by V1
NOT USED
Attribute 1 [V1]
Attribute 1 [V0]
Delta appended to Attribute 1 [V0]
Updated by Component B when backing-up Attribute 1
Updated by Component B when backing-up Attribute 1

Figure 2.3: Upgrading to attribute increased in size

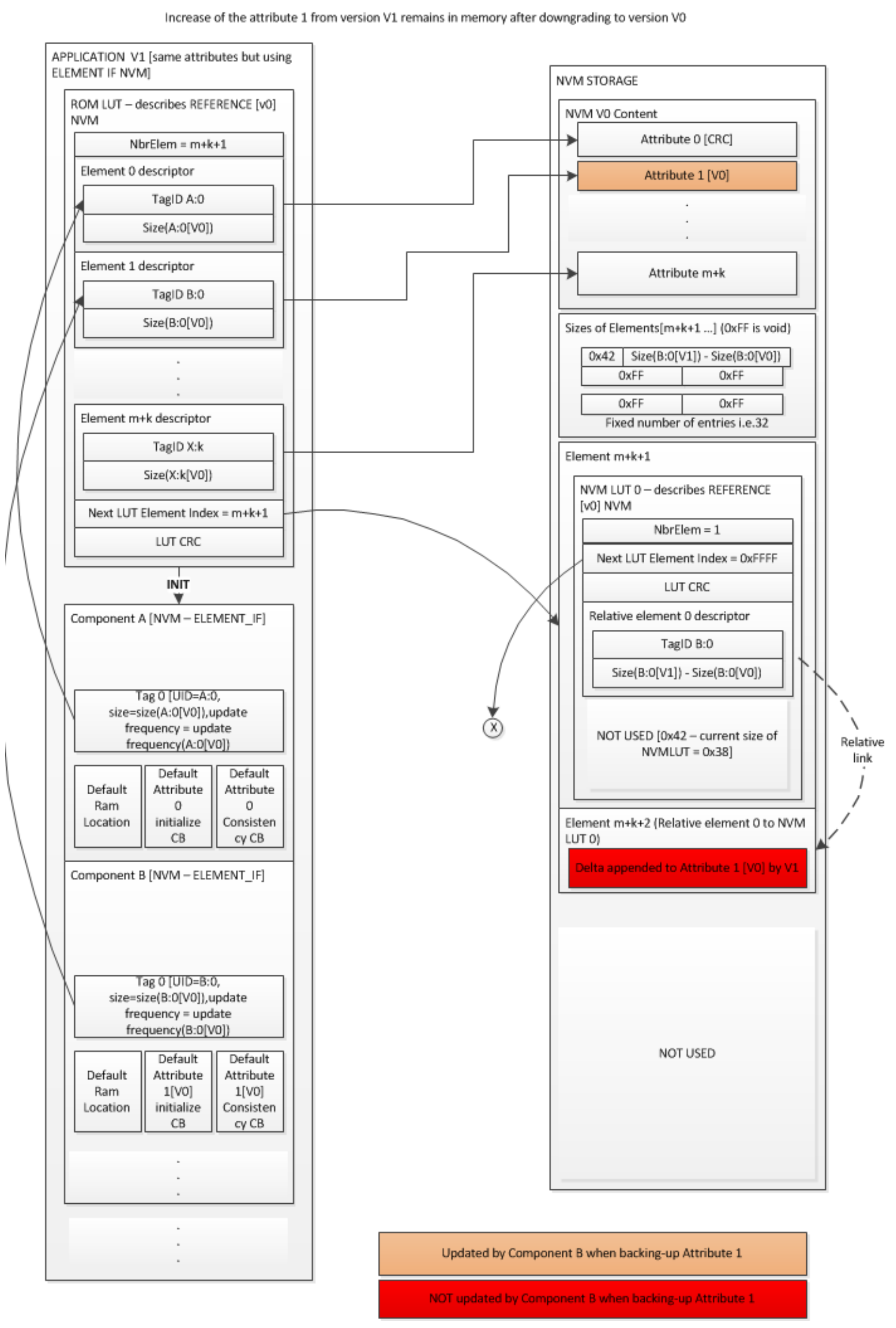

Increase of the attribute 1 from version V1 remains in memory after downgrading to version V0
APPLICATION V1 [same attributes but using ELEMENT IF NVM]
ROM LUT – describes REFERENCE [v0] NVM
NbrElem = m+k+1
Element 0 descriptor
TagID A:0
Size(A:0[V0])
Element 1 descriptor
TagID B:0
Size(B:0[V0])
Element m+k descriptor
TagID X:k
Size(X:k[V0])
Next LUT Element Index = m+k+1
LUT CRC
INIT
Component A [NVM – ELEMENT_IF]
Tag 0 [UID=A:0, size=size(A:0[V0]),update frequency = update frequency(A:0[V0])]
Default Ram Location
Default Attribute 0 initialize CB
Default Attribute 0 Consistency CB
Component B [NVM – ELEMENT_IF]
Tag 0 [UID=B:0, size=size(B:0[V0]),update frequency = update frequency(B:0[V0])]
Default Ram Location
Default Attribute 1[V0] initialize CB
Default Attribute 1[V0] Consistency CB
NVM STORAGE
NVM V0 Content
Attribute 0 [CRC]
Attribute 1 [V0]
Attribute m+k
Sizes of Elements[m+k+1 ...] (0xFF is void)
0x42
Size(B:0[V1]) - Size(B:0[V0])
0xFF
0xFF
0xFF
0xFF
Fixed number of entries i.e.32
Element m+k+1
NVM LUT 0 – describes REFERENCE [v0] NVM
NbrElem = 1
Next LUT Element Index = 0xFFFF
LUT CRC
Relative element 0 descriptor
TagID B:0
Size(B:0[V1]) - Size(B:0[V0])
NOT USED [0x42 – current size of NVMLUT = 0x38]
X
Relative link
Element m+k+2 (Relative element 0 to NVM LUT 0)
Delta appended to Attribute 1 [V0] by V1
NOT USED
Updated by Component B when backing-up Attribute 1
NOT updated by Component B when backing-up Attribute 1

Figure 2.4: Downgrading to attribute smaller in size

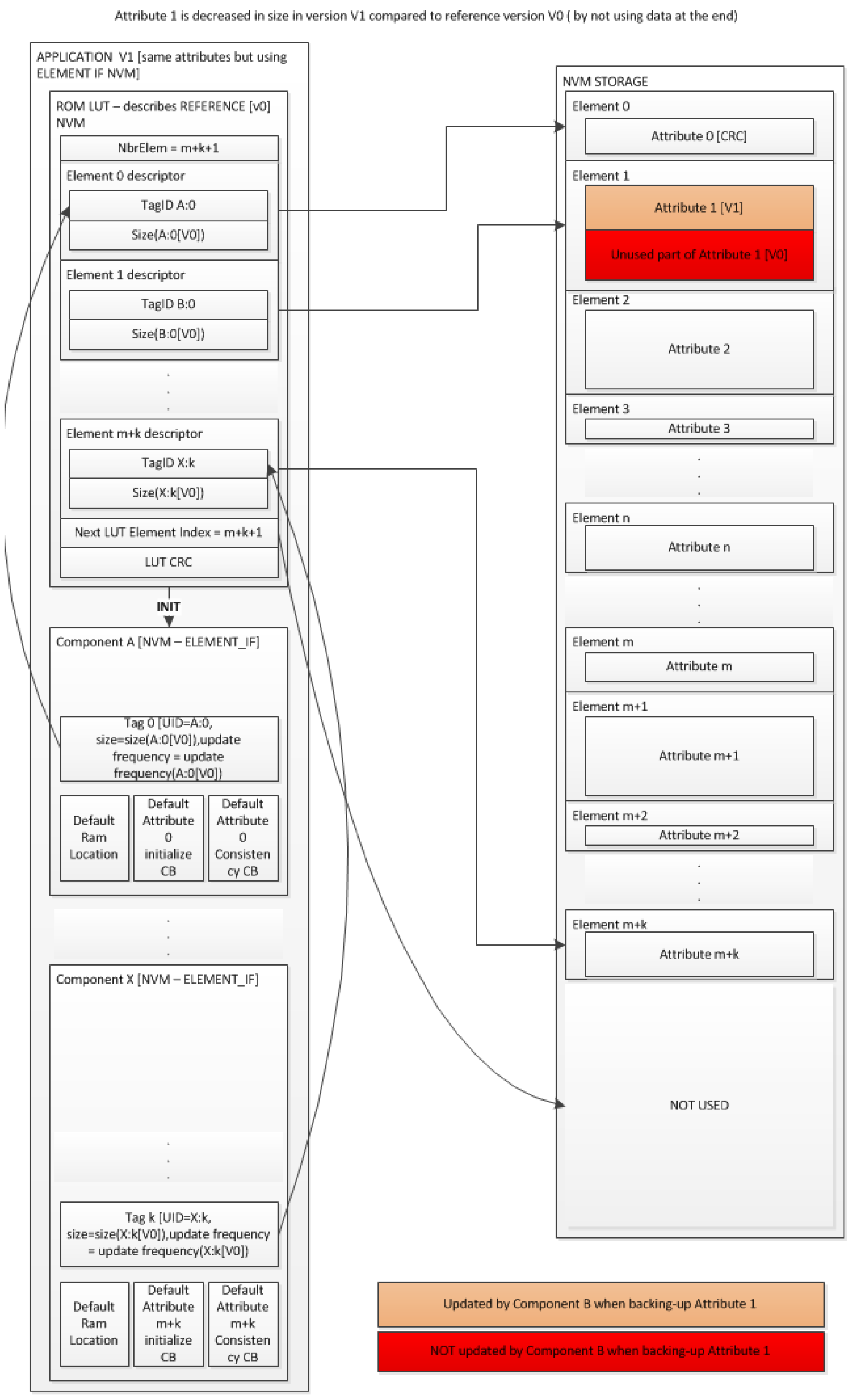

Attribute 1 is decreased in size in version V1 compared to reference version V0 (by not using data at the end)
APPLICATION V1 [same attributes but using ELEMENT IF NVM]
ROM LUT – describes REFERENCE [v0] NVM
NbrElem = m+k+1
Element 0 descriptor
TagID A:0
Size(A:0[V0])
Element 1 descriptor
TagID B:0
Size(B:0[V0])
Element m+k descriptor
TagID X:k
Size(X:k[V0])
Next LUT Element Index = m+k+1
LUT CRC
INIT
Component A [NVM – ELEMENT_IF]
Tag 0 [UID=A:0, size=size(A:0[V0]),update frequency = update frequency(A:0[V0])
Default Ram Location
Default Attribute 0 initialize CB
Default Attribute 0 Consistency CB
Component X [NVM – ELEMENT_IF]
Tag k [UID=X:k, size=size(X:k[V0]),update frequency = update frequency(X:k[V0])
Default Ram Location
Default Attribute m+k initialize CB
Default Attribute m+k Consistency CB
NVM STORAGE
Element 0
Attribute 0 [CRC]
Element 1
Attribute 1 [V1]
Unused part of Attribute 1 [V0]
Element 2
Attribute 2
Element 3
Attribute 3
Element n
Attribute n
Element m
Attribute m
Element m+1
Attribute m+1
Element m+2
Attribute m+2
Element m+k
Attribute m+k
NOT USED
Updated by Component B when backing-up Attribute 1
NOT updated by Component B when backing-up Attribute 1

Figure 2.5: Upgrading to attribute decreased in size

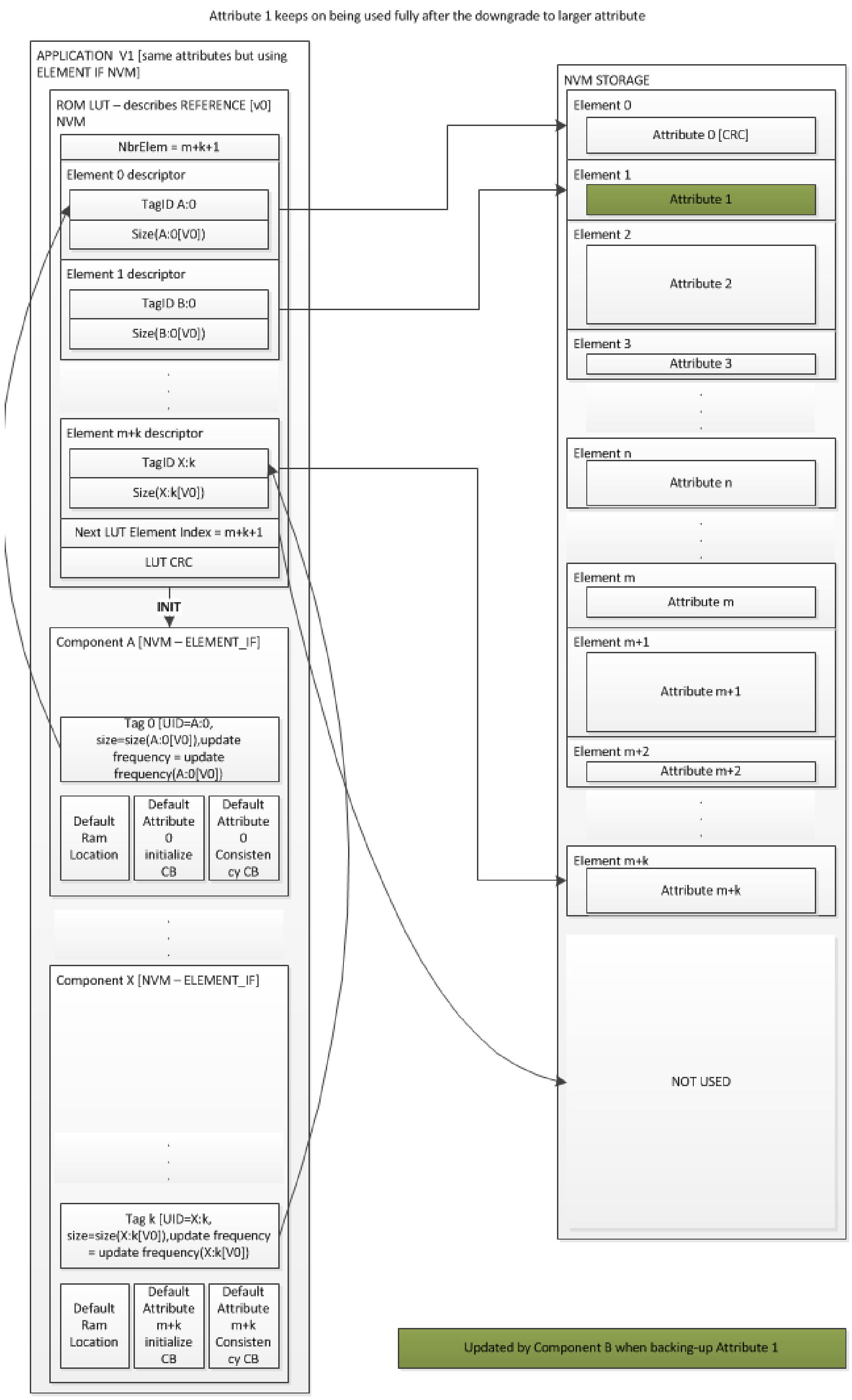

Attribute 1 keeps on being used fully after the downgrade to larger attribute
APPLICATION V1 [same attributes but using ELEMENT IF NVM]
ROM LUT – describes REFERENCE [v0] NVM
NbrElem = m+k+1
Element 0 descriptor
TagID A:0
Size(A:0[V0])
Element 1 descriptor
TagID B:0
Size(B:0[V0])
Element m+k descriptor
TagID X:k
Size(X:k[V0])
Next LUT Element Index = m+k+1
LUT CRC
INIT
Component A [NVM – ELEMENT_IF]
Tag 0 [UID=A:0, size=size(A:0[V0]),update frequency = update frequency(A:0[V0])
Default Ram Location
Default Attribute 0 initialize CB
Default Attribute 0 Consistency CB
Component X [NVM – ELEMENT_IF]
Tag k [UID=X:k, size=size(X:k[V0]),update frequency = update frequency(X:k[V0])
Default Ram Location
Default Attribute m+k initialize CB
Default Attribute m+k Consistency CB
NVM STORAGE
Element 0
Attribute 0 [CRC]
Element 1
Attribute 1
Element 2
Attribute 2
Element 3
Attribute 3
Element n
Attribute n
Element m
Attribute m
Element m+1
Attribute m+1
Element m+2
Attribute m+2
Element m+k
Attribute m+k
NOT USED
Updated by Component B when backing-up Attribute 1

Figure 2.6: Downgrading to attribute larger in size

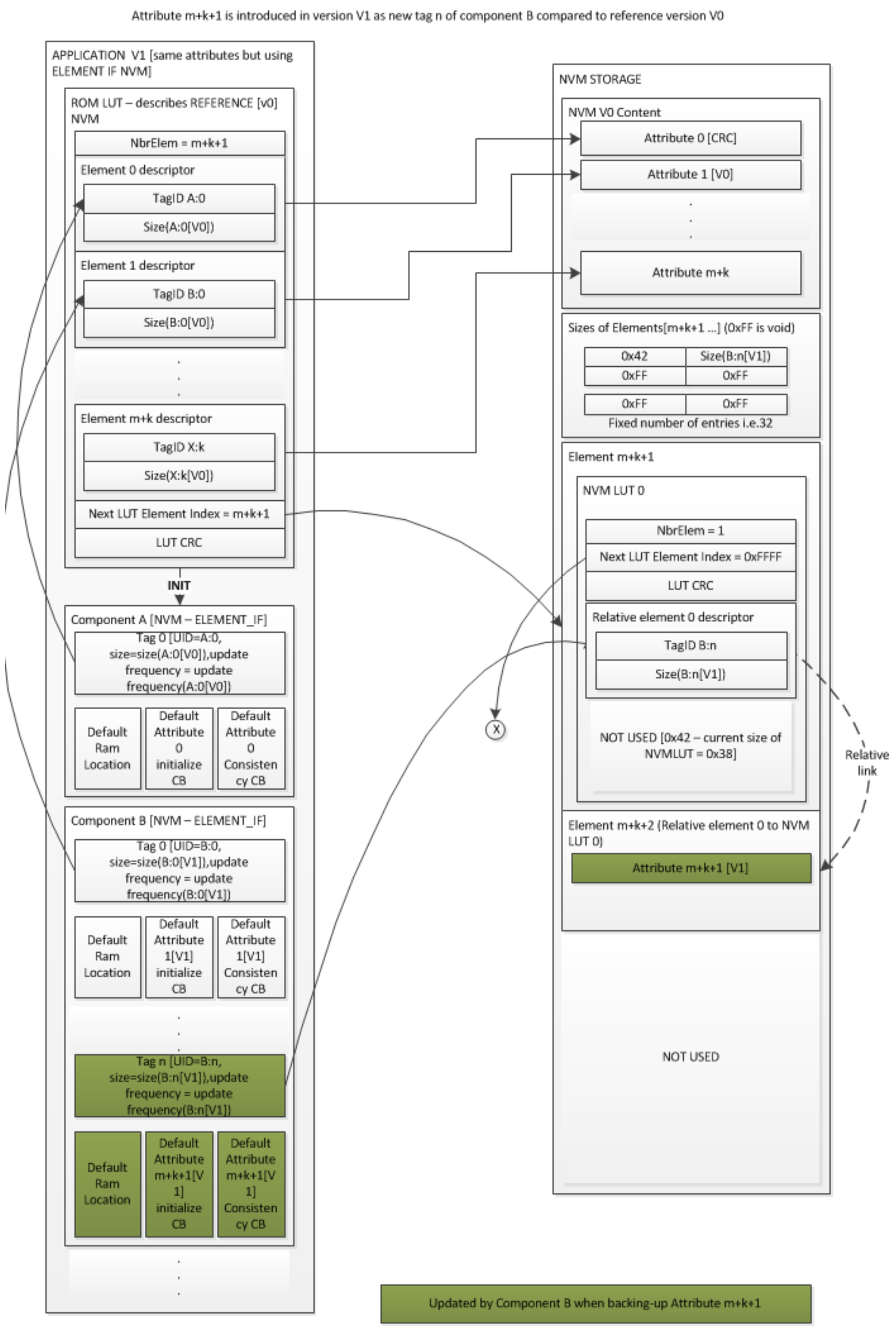
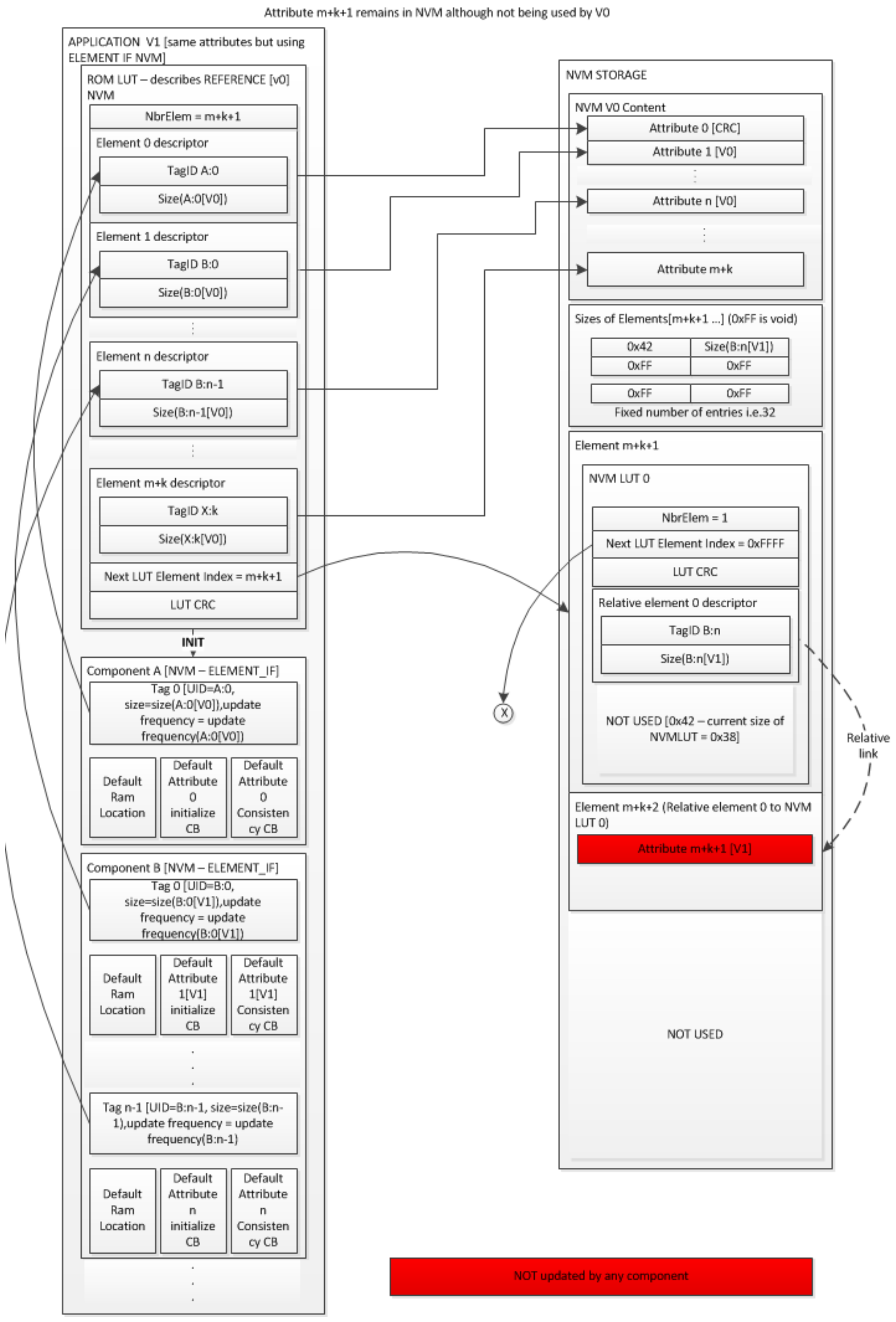

Attribute m+k+1 is introduced in version V1 as new tag n of component B compared to reference version V0
APPLICATION V1 [same attributes but using ELEMENT IF NVM]
ROM LUT – describes REFERENCE [v0] NVM
NbrElem = m+k+1
Element 0 descriptor
TagID A:0
Size(A:0[V0])
Element 1 descriptor
TagID B:0
Size(B:0[V0])
Element m+k descriptor
TagID X:k
Size(X:k[V0])
Next LUT Element Index = m+k+1
LUT CRC
INIT
Component A [NVM – ELEMENT_IF]
Tag 0 [UID=A:0, size=size(A:0[V0]),update frequency = update frequency(A:0[V0])
Default Ram Location
Default Attribute 0 initialize CB
Default Attribute 0 Consistency CB
Component B [NVM – ELEMENT_IF]
Tag 0 [UID=B:0, size=size(B:0[V1]),update frequency = update frequency(B:0[V1])
Default Ram Location
Default Attribute 1[V1] initialize CB
Default Attribute 1[V1] Consistency CB
Tag n [UID=B:n, size=size(B:n[V1]),update frequency = update frequency(B:n[V1])
Default Ram Location
Default Attribute m+k+1[V1] initialize CB
Default Attribute m+k+1[V1] Consistency CB
NVM STORAGE
NVM V0 Content
Attribute 0 [CRC]
Attribute 1 [V0]
Attribute m+k
Sizes of Elements[m+k+1 ...] (0xFF is void)
0x42
Size(B:n[V1])
0xFF
0xFF
0xFF
0xFF
Fixed number of entries i.e.32
Element m+k+1
NVM LUT 0
NbrElem = 1
Next LUT Element Index = 0xFFFF
LUT CRC
Relative element 0 descriptor
TagID B:n
Size(B:n[V1])
NOT USED [0x42 – current size of NVMLUT = 0x38]
X
Relative link
Element m+k+2 (Relative element 0 to NVM LUT 0)
Attribute m+k+1 [V1]
NOT USED
Updated by Component B when backing-up Attribute m+k+1

Figure 2.7: Upgrading by adding attributes



Attribute m+k+1 remains in NVM although not being used by V0
APPLICATION V1 [same attributes but using ELEMENT IF NVM]
ROM LUT – describes REFERENCE [v0] NVM
NbrElem = m+k+1
Element 0 descriptor
TagID A:0
Size(A:0[V0])
Element 1 descriptor
TagID B:0
Size(B:0[V0])
Element n descriptor
TagID B:n-1
Size(B:n-1[V0])
Element m+k descriptor
TagID X:k
Size(X:k[V0])
Next LUT Element Index = m+k+1
LUT CRC
INIT
Component A [NVM – ELEMENT_IF]
Tag 0 [UID=A:0, size=size(A:0[V0]),update frequency = update frequency(A:0[V0])
Default Ram Location
Default Attribute 0 initialize CB
Default Attribute 0 Consistency CB
Component B [NVM – ELEMENT_IF]
Tag 0 [UID=B:0, size=size(B:0[V1]),update frequency = update frequency(B:0[V1])
Default Ram Location
Default Attribute 1[V1] initialize CB
Default Attribute 1[V1] Consistency CB
Tag n-1 [UID=B:n-1, size=size(B:n-1),update frequency = update frequency(B:n-1)
Default Ram Location
Default Attribute n initialize CB
Default Attribute n Consistency CB
NVM STORAGE
NVM V0 Content
Attribute 0 [CRC]
Attribute 1 [V0]
Attribute n [V0]
Attribute m+k
Sizes of Elements[m+k+1 ...] (0xFF is void)
0x42
Size(B:n[V1])
0xFF
0xFF
0xFF
0xFF
Fixed number of entries i.e.32
Element m+k+1
NVM LUT 0
NbrElem = 1
Next LUT Element Index = 0xFFFF
LUT CRC
Relative element 0 descriptor
TagID B:n
Size(B:n[V1])
NOT USED [0x42 – current size of NVMLUT = 0x38]
Element m+k+2 (Relative element 0 to NVM LUT 0)
Attribute m+k+1 [V1]
Relative link
NOT USED
X
NOT updated by any component

Figure 2.8: Downgrading to smaller number of attributes

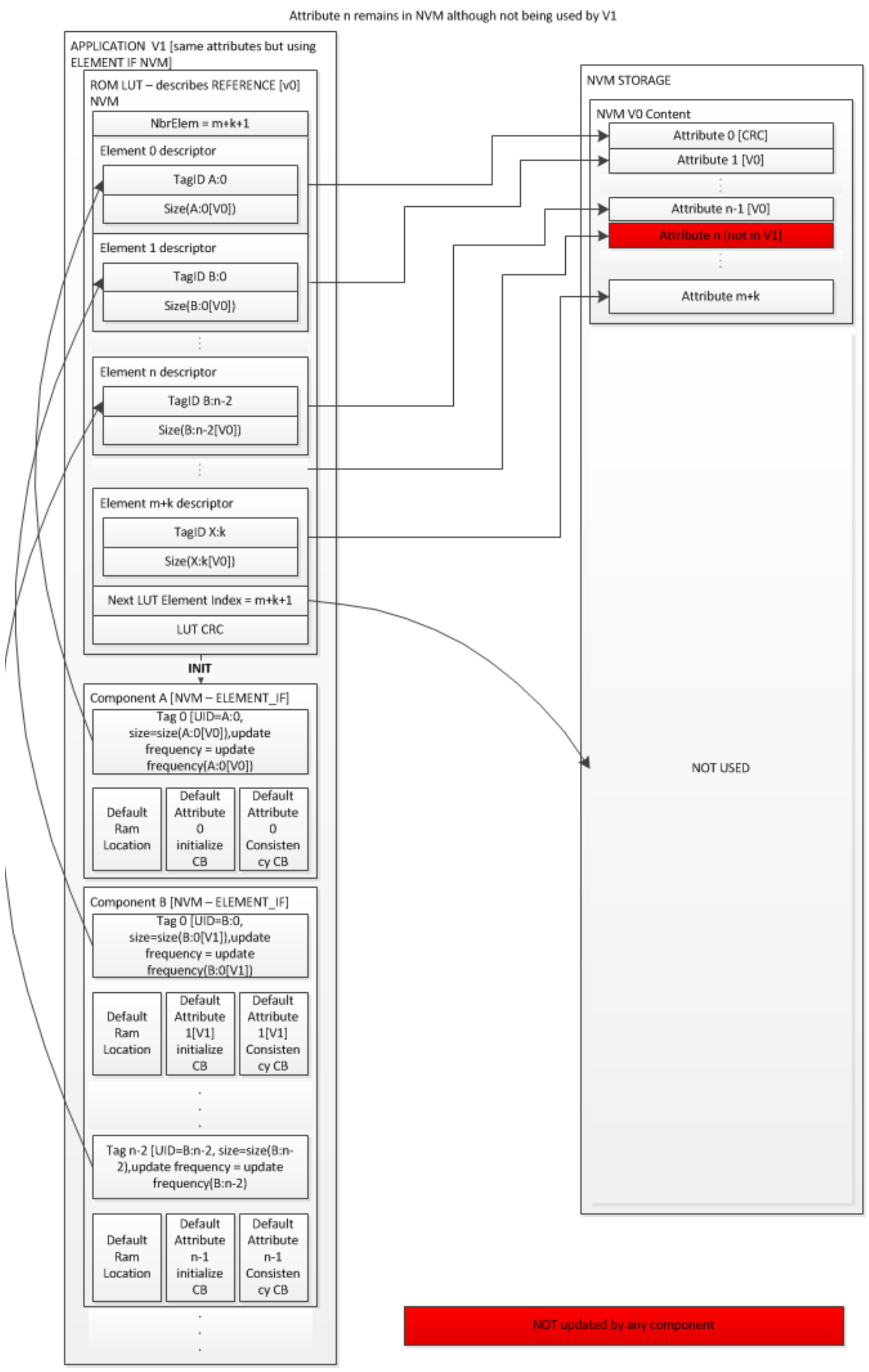

Attribute n remains in NVM although not being used by V1
APPLICATION V1 [same attributes but using ELEMENT IF NVM]
ROM LUT – describes REFERENCE [v0] NVM
NbrElem = m+k+1
Element 0 descriptor
TagID A:0
Size(A:0[V0])
Element 1 descriptor
TagID B:0
Size(B:0[V0])
Element n descriptor
TagID B:n-2
Size(B:n-2[V0])
Element m+k descriptor
TagID X:k
Size(X:k[V0])
Next LUT Element Index = m+k+1
LUT CRC
INIT
Component A [NVM – ELEMENT_IF]
Tag 0 [UID=A:0, size=size(A:0[V0]),update frequency = update frequency(A:0[V0])
Default Ram Location
Default Attribute 0 initialize CB
Default Attribute 0 Consistency CB
Component B [NVM – ELEMENT_IF]
Tag 0 [UID=B:0, size=size(B:0[V1]),update frequency = update frequency(B:0[V1])
Default Ram Location
Default Attribute 1[V1] initialize CB
Default Attribute 1[V1] Consistency CB
Tag n-2 [UID=B:n-2, size=size(B:n-2),update frequency = update frequency(B:n-2)
Default Ram Location
Default Attribute n-1 initialize CB
Default Attribute n-1 Consistency CB
NVM STORAGE
NVM V0 Content
Attribute 0 [CRC]
Attribute 1 [V0]
Attribute n-1 [V0]
Attribute n [not in V1]
Attribute m+k
NOT USED
NOT updated by any component

Figure 2.9: Upgrading to smaller number of attributes

Elements themselves are defined in the NVM in two ways. The inherited NVM has implicit definition of the elements as inherited attributes and their sizes. The subsequent elements are added in the NVM in a platform specific way. Simple way to implement platform specific element access is to have a table of fixed size located right after the inherited NVM. This table can contain array of values (8 or 16 bit) specifying element sizes. It should be initialized to all 1 values (all

0 and all 1 values indicating void element). Elements would be added one after another and the size array would be updated accordingly. When locating the element within the NVM, first query would be made to program memory look-up table. If element is not found there, the NVM element size array located right after the last known inherited attribute could be used to locate noninherited elements.

The explained manipulations with NVM based look-up tables are illustrated in the following images:

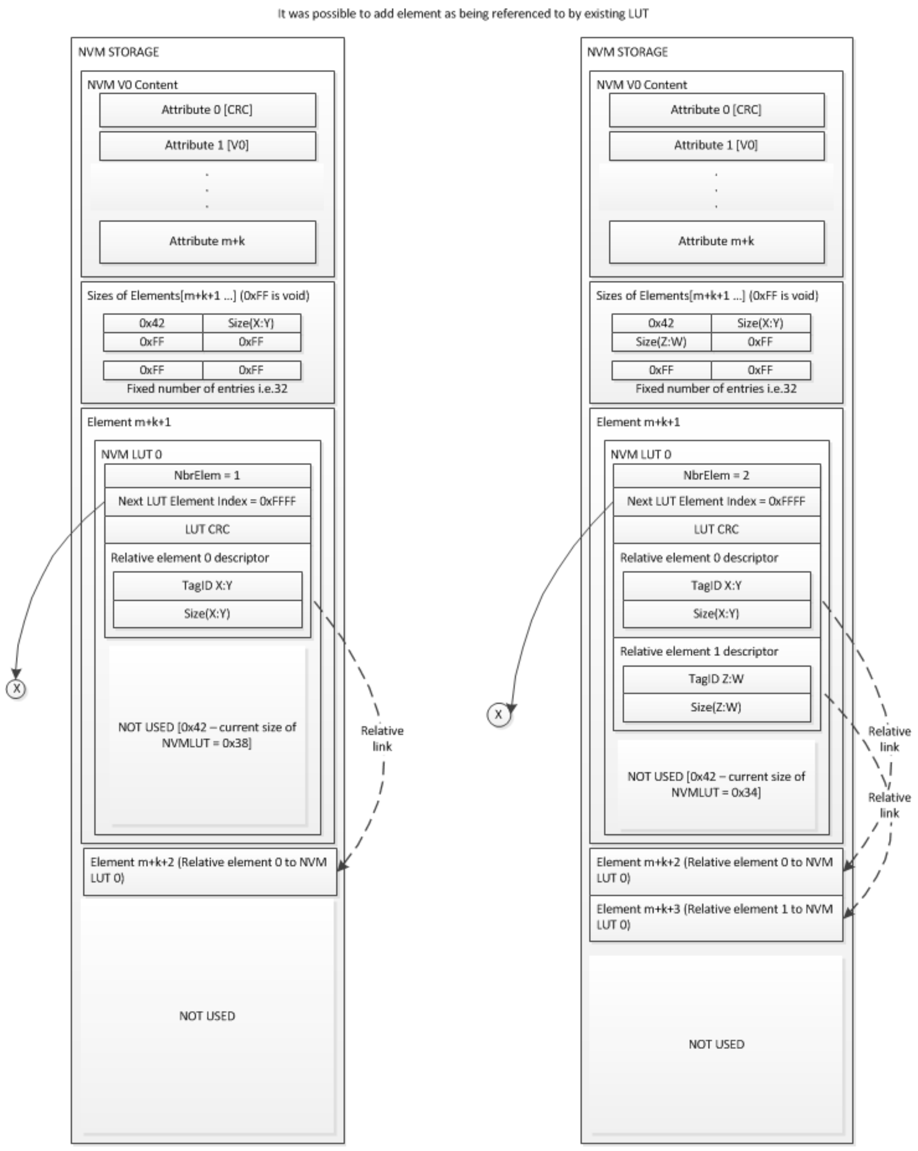

It was possible to add element as being referenced to by existing LUT
NVM STORAGE
NVM V0 Content
Attribute 0 [CRC]
Attribute 1 [V0]
Attribute m+k
Sizes of Elements[m+k+1 ...] (0xFF is void)
0x42
Size(X:Y)
0xFF
0xFF
0xFF
0xFF
Fixed number of entries i.e.32
Element m+k+1
NVM LUT 0
NbrElem = 1
Next LUT Element Index = 0xFFFF
LUT CRC
Relative element 0 descriptor
TagID X:Y
Size(X:Y)
NOT USED [0x42 – current size of NVMLUT = 0x38]
Relative link
X
Element m+k+2 (Relative element 0 to NVM LUT 0)
NOT USED
NVM STORAGE
NVM V0 Content
Attribute 0 [CRC]
Attribute 1 [V0]
Attribute m+k
Sizes of Elements[m+k+1 ...] (0xFF is void)
0x42
Size(X:Y)
Size(Z:W)
0xFF
0xFF
0xFF
Fixed number of entries i.e.32
Element m+k+1
NVM LUT 0
NbrElem = 2
Next LUT Element Index = 0xFFFF
LUT CRC
Relative element 0 descriptor
TagID X:Y
Size(X:Y)
Relative element 1 descriptor
TagID Z:W
Size(Z:W)
NOT USED [0x42 – current size of NVMLUT = 0x34]
Relative link
Relative link
X
Element m+k+2 (Relative element 0 to NVM LUT 0)
Element m+k+3 (Relative element 1 to NVM LUT 0)
NOT USED

Figure 2.10: Extending the existing NVM LUT with additional element

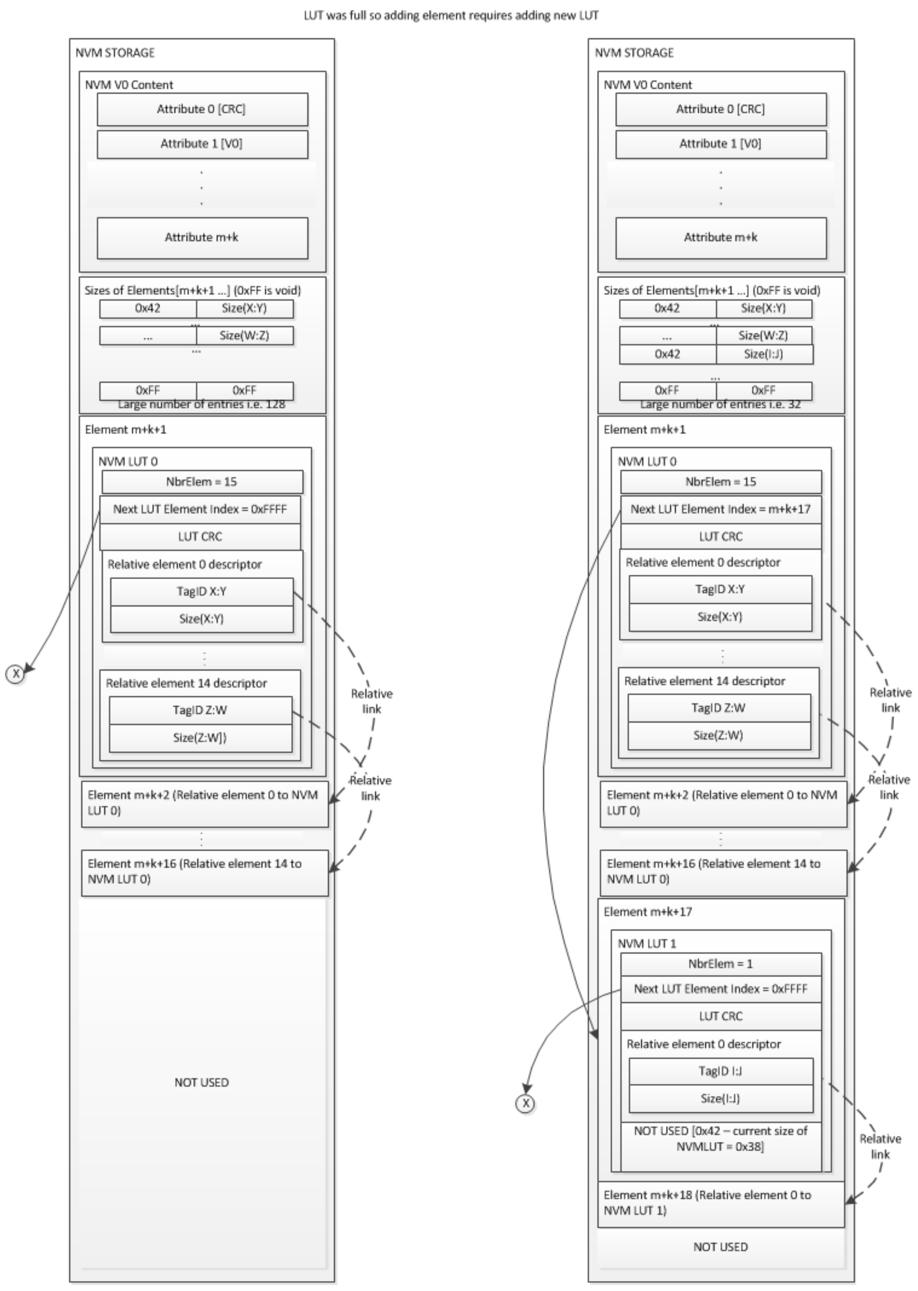

LUT was full so adding element requires adding new LUT
NVM STORAGE
NVM V0 Content
Attribute 0 [CRC]
Attribute 1 [V0]
Attribute m+k
Sizes of Elements[m+k+1 ...] (0xFF is void)
0x42
Size(X:Y)
...
Size(W:Z)
0xFF
Large number of entries i.e. 128
Element m+k+1
NVM LUT 0
NbrElem = 15
Next LUT Element Index = 0xFFFF
LUT CRC
Relative element 0 descriptor
TagID X:Y
Size(X:Y)
Relative element 14 descriptor
TagID Z:W
Size(Z:W])
Relative link
Element m+k+2 (Relative element 0 to NVM LUT 0)
Element m+k+16 (Relative element 14 to NVM LUT 0)
NOT USED
X
NVM STORAGE
NVM V0 Content
Attribute 0 [CRC]
Attribute 1 [V0]
Attribute m+k
Sizes of Elements[m+k+1 ...] (0xFF is void)
0x42
Size(X:Y)
...
Size(W:Z)
0x42
Size(I:J)
0xFF
Large number of entries i.e. 32
Element m+k+1
NVM LUT 0
NbrElem = 15
Next LUT Element Index = m+k+17
LUT CRC
Relative element 0 descriptor
TagID X:Y
Size(X:Y)
Relative element 14 descriptor
TagID Z:W
Size(Z:W)
Relative link
Element m+k+2 (Relative element 0 to NVM LUT 0)
Element m+k+16 (Relative element 14 to NVM LUT 0)
Element m+k+17
NVM LUT 1
NbrElem = 1
Next LUT Element Index = 0xFFFF
LUT CRC
Relative element 0 descriptor
TagID I:J
Size(I:J)
NOT USED [0x42 – current size of NVMLUT = 0x38]
Element m+k+18 (Relative element 0 to NVM LUT 1)
NOT USED
X

Figure 2.11: Adding NVM LUT with refering to newly added element

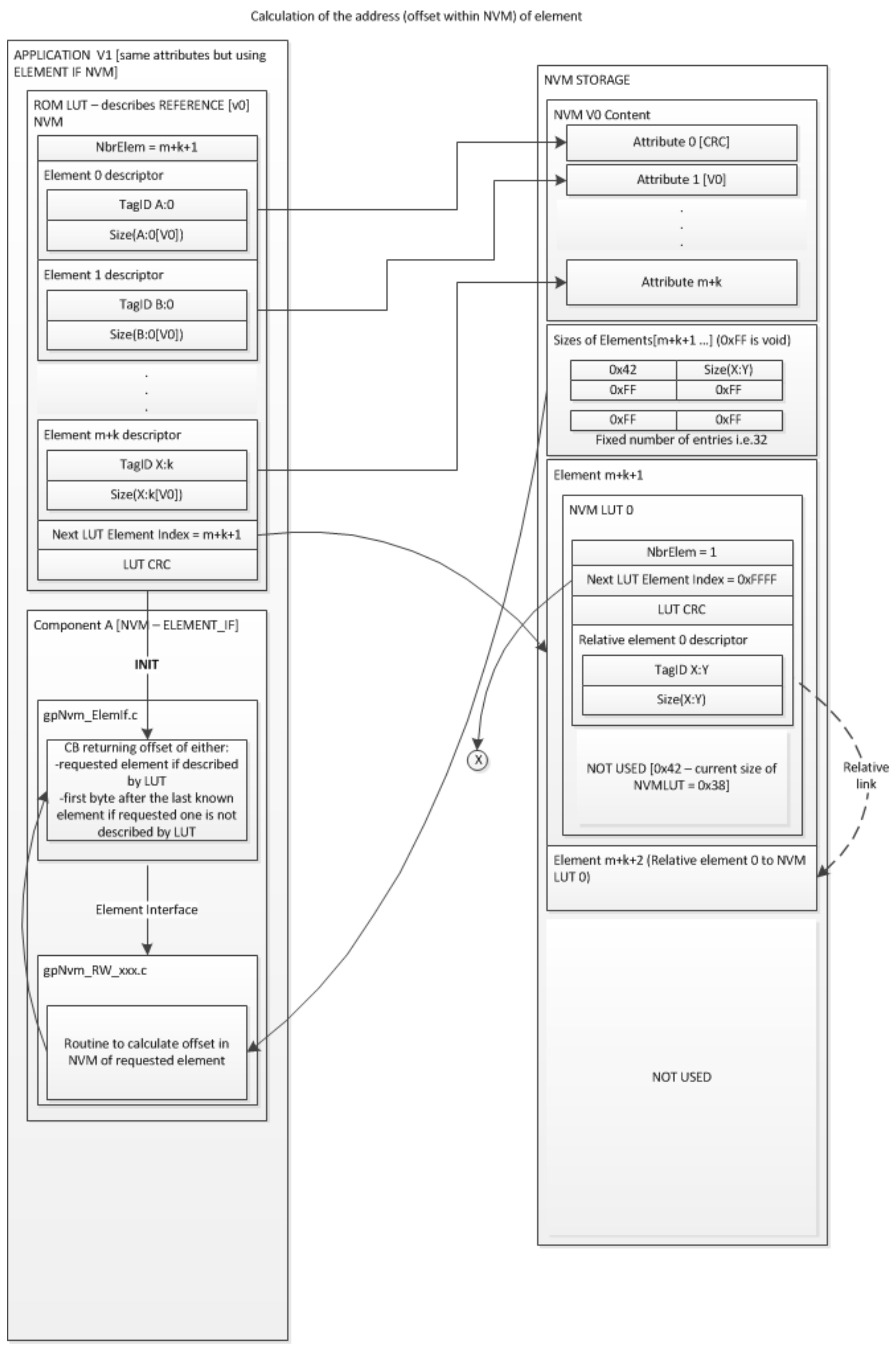

Calculation of the address (offset within NVM) of element
APPLICATION V1 [same attributes but using ELEMENT IF NVM]
ROM LUT – describes REFERENCE [v0] NVM
NbrElem = m+k+1
Element 0 descriptor
TagID A:0
Size(A:0[V0])
Element 1 descriptor
TagID B:0
Size(B:0[V0])
Element m+k descriptor
TagID X:k
Size(X:k[V0])
Next LUT Element Index = m+k+1
LUT CRC
Component A [NVM – ELEMENT_IF]
INIT
gpNvm_ElemIf.c
CB returning offset of either: -requested element if described by LUT -first byte after the last known element if requested one is not described by LUT
Element Interface
gpNvm_RW_xxx.c
Routine to calculate offset in NVM of requested element
NVM STORAGE
NVM V0 Content
Attribute 0 [CRC]
Attribute 1 [V0]
Attribute m+k
Sizes of Elements[m+k+1 ...] (0xFF is void)
0x42
Size(X:Y)
0xFF
0xFF
0xFF
0xFF
Fixed number of entries i.e.32
Element m+k+1
NVM LUT 0
NbrElem = 1
Next LUT Element Index = 0xFFFF
LUT CRC
Relative element 0 descriptor
TagID X:Y
Size(X:Y)
NOT USED [0x42 – current size of NVMLUT = 0x38]
X
Relative link
Element m+k+2 (Relative element 0 to NVM LUT 0)
NOT USED

Figure 2.12: Calculation of the address of the element in the NVM

All of the premises for the driver stated so far imply that the way the attributes are registered to the NVM driver had to change. Information about attribute identity has to be provided explicitly rather than implicitly so new registration function is introduced. Elements may be registered individualy or in group(s) where order (IDs within componen or componets them selves) is not relevant. Any mix is possible, match will be found. Since the element tags had to change in structure to accomodate this explicit information, opportunity is taken to also change the way the default value initialization is done. Instead of providing the pointer to default data (if any) a default initialization function reference is provided making large sparse default information compressed beter in the program memory. Another addition to the tag is element (and not section) specific consistency checker function reference that can be used to asses integrity per attribute.

The special attention is given to make the best effort approach in reusing the attribute values when traversing among the firmware versions. The standard applications have the first attribute contain the CRC signing the section-attribute structure of the NVM. The element interface based applications will reconstruct this information from the program memory look-up table and will be able to asses if the inherited NVM is valid. This information is not dependant on the actual attributes used by the element interface based application. All of the reused attributes are updated in the inherited memory and if some attributes are extended, only the delta of the extended attribute will be allocated in the separate NVM element.. Initial value of extended attribute will be composite of already present value in inherited attribute and the rest will be default initialized. In unlikely event of reducing the inherited attribute size, the part of the inherited arribute known and used by the new application will be kept up to date. IMPORTANT NOTE: although in most of cases this approach provides valid attribute data when up or downgrading, care must be taken that the attribute values created by default extension of previous shorter attributes or by truncating previous longer attributes actually make sense. It is recommended to use consistency checker functions if the consistency can be broken by up/downgrading.

Compatibility between the applications in regard to NVM is defined by two requirements:

- All mutually compatible versions have to use same base program memory LUT content
- All mutually compatible versions have to use same unique IDs for same attributes

One special situation occurs with applications offering remote NVM service over the serial interface. The RW_NVM storage is allocated as array of elements that total in size required remote NVM. These elements are attributes being registered with common tag ID but internally their ID is incremented on every registration of such special element to stop them from being treated as one single attribute.

2.2.2 Function Documentation

gpNvm_Config()

```
void gpNvm_Config (
        const ROM gpNvm_LookUpTableHeader_t * baseTableHeader,
        const ROM gpNvm_ElementDefine_t * baseTableElements,
        const ROM gpNvm_VersionCrc_t * inheritedNvmVersionCrc )
```

This function call may be used if there can be different possibilities for the NVM structure inherited. This function can't be used for detection of the type of the NVM used since the failure to validate NVM content against the registered table causes NVM defaulting.

Parameters

baseTableHeader	A reference to the header of the base LUT
baseTableElements	A reference to the array of element descriptors of the base LUT

gpNvm_RegisterElement()

```
void gpNvm_RegisterElement (
                const ROM gpNvm_IdentifiableTag_t * pTag )
```

Registers an element to the NVM software block. The function should be called from the _init function of the component for each element being registered.

Parameters

pTag	A pointer to a tag of the element allocated in flash contining the information about the attribute.

gpNvm_RegisterElements()

```
void gpNvm_RegisterElements (
                const ROM gpNvm_IdentifiableTag_t * pTag,
                UInt8 nbrOfTags )
```

Registers multiple of elements reported as array of tags to the NVM software block. The function should be called from the _init function of the component for each element being registered.

Parameters

pTag	A pointer to array of tags of the elements allocated in flash contining the information about the attributes.

Chapter 3

Data Structure Documentation

3.1 gpNvm_CompatibilityEntry Struct Reference

The gpNvm_CompatibilityEntry structure specifies the information for compatible NVM versions.

3.2 gpNvm_CompatibilityEntry_t Struct Reference

Data Fields

- UInt16 **crc**
- gpNvm_cbUpdate_t **upgradeCb**
- gpNvm_cbUpdate_t **downgradeCb**

3.3 gpNvm_ElementDefine_t Struct Reference

Data Fields

- UInt16 uniqueTagId

  Unique tag identifier composed from the unique component ID located in MSB and unique tag ID within the component located in LSB.
- UInt16 elementSize

  The number of bytes of the attribute in the element.

3.4 gpNvm_IdentifiableTag Struct Reference

The gpNvm_IdentifiableTag structure specifies the fields that describe an uniquely identifiable nvm-tag.

3.5 gpNvm_IdentifiableTag_t Struct Reference

Data Fields

- UInt16 uniqueTagId

  Unique tag identifier composed from the unique component ID located in MSB and unique tag ID within the component located in LSB.

- UInt8 ∗ pRamLocation

  A pointer to the corresponding RAM memory of the attribute to be backuped in nvm.
- UInt16 size

  The number of bytes of the nvm-tag.
- gpNvm_UpdateFrequency_t updateFrequency

  An indication of the update frequency.
- Bool(∗ DefaultValueInitializerCB)(const ROM void ∗pTag, UInt8 ∗pBuffer)

  A pointer to function that initializes attribute default values. 0xFF will be used as default values if pointer is specified as NULL.
- Bool(∗ ConsistencyCheckerCB)(const ROM void ∗pTag)

  A pointer to function that performs consistency check of the attribute value.

3.6 gpNvm_LookUpTableHeader Struct Reference

The gpNvm_LookUpTableHeader structure specifies the fields that describe header of the element LookUp Table.

3.7 gpNvm_LookUpTableHeader_t Struct Reference

Data Fields

- UInt16 nbrTags

  The number of tags for attribute elements contained in the LookUp Table.
- UInt16 indexNextLUT

  The index of the element containing next LookUp Table. LUTs are forming single direction linked list.
- gpNvm_VersionCrc_t crcLUT

  The CRC calculated over the rest of the header and element defines belonging to the LookUp Table.

3.8 gpNvm_Section_t Struct Reference

Data Fields

- UInt8 **componentId**
- const ROM gpNvm_Tag_t ∗FLASH_PROGMEM **tags**
- UInt8 **nbrOfTags**
- UInt8 ∗ **baseAddr**
- gpNvm_cbCheckConsistency_t **cbCheckConsistency**

3.9 gpNvm_Tag_t Struct Reference

The gpNvm_Tag structure specifies the fields that describe an nvm-tag.

Data Fields

- UInt8 ∗ pRamLocation

    A pointer to the corresponding RAM memory of the attributed to be backuped in nvm.
- UInt16 size

    The number of bytes of the nvm-tag.
- gpNvm_UpdateFrequency_t updateFrequency

    An indication of the update frequency.
- GP_NVM_CONST ROM UInt8 ∗ pDefaultValues

    A pointer to flash memory containing the default values for the tag, 0xFF will be used as default values if pointer is specified as NULL.

Chapter 4

File Documentation

4.1 gpNvm.h File Reference

Data Structures

- struct gpNvm_Tag_t

  The gpNvm_Tag structure specifies the fields that describe an nvm-tag.
- struct gpNvm_Section_t
- struct gpNvm_CompatibilityEntry_t

Macros

- #define **gpNvm_LookupTable_Handle_Invalid** (0xff)
- #define gpNvm_AllComponents 0xEE

  Wildcard used to select all nvm-sections in a backup/restore/clear call.
- #define gpNvm_AllTags 0xEE

  Wildcard used to select all tags of the specified nvm-sections in a backup/restore/clear call.
- #define GP_NVM_SECTIONID_TAG 16
- #define GP_NVM_MAX_TOKENLENGTH 1
- #define GP_NVM_MAX_PAYLOADLENGTH 241
- #define **GP_NVM_NBR_OF_POOLS** 1
- #define **GP_NVM_NBR_OF_UNIQUE_TAGS** 255

Typedefs

- typedef UInt8 **gpNvm_LookupTable_Handle_t**
- typedef UInt16 gpNvm_Type_t

  defined(GP_NVM_NBR_OF_POOLS)
- typedef UInt8 **gpNvm_KeyIndex_t**
- typedef UInt16 **gpNvm_VersionCrc_t**
- typedef Bool(∗ gpNvm_cbCheckConsistency_t) (void)

  The gpNvm_cbCheckConsistency_t defines the callback type for an additional section specific check-consistency routine (see gpNvm_RegisterSection())
- typedef Bool(∗ gpNvm_cbUpdate_t) (void)

  The gpNvm_cbUpdate_t defines the callback type for the up or downgrade routine between NVM versions.

Functions

- void gpNvm_Init (void)

  Initializes the NVM software block. After execution of this function data can be read or written to NVM.
- void **gpNvm_DeInit** (void)
- UInt16 gpNvm_ReserveRemoteNvmSpace (UInt16 requiredSize)

  Reserves part of space on local NVM for remote NVM usage. This function should be called at most once. If not called, no space is reserved for remote NVM.
- void gpNvm_ClearNvm (void)

  Clears the complete NVM. This function clears the complete NVM. This includes all tags of all registered nvm-sections and all legacy implementations of the component specific interfaces described in the other sections. Readonly tags will not be erased unless the NVM was unlocked using gpNvm_SetLock().
- void gpNvm_Flush (void)

  Requests a forced flush of the write buffers to NVM. This function request the nvm component to flush all pending write data to NVM. If the NVM implementation doesn't use buffers, the function should be stubbed.
- void gpNvm_Dump (UInt8 componentId, UInt8 tagId)

  Dumps the NVM content. This function dumps the content of the specified tag of the specified component. It will also indicate with a '∗' that this content is not in sync with the current values of the corresponding ram variables.
- void gpNvm_DumpStructure (void)

  This function dumps the structure of the NVM tags and NVM CRC.
- void gpNvm_Backup (UInt8 componentId, UInt8 tagId, UInt8 ∗pRamLocation)

  Backup the content of the specified NVM-tag. The content of the RAM memory (pRamLocation parameter) specified by the combination of the component and tag will be written to the non-volitale memory.
- void gpNvm_Restore (UInt8 componentId, UInt8 tagId, UInt8 ∗pRamLocation)

  Restore the content of the specified NVM-tag using the provided RAM location. The content of the RAM memory (pRamLocation parameter) specified by the combination of the component and tag will be updated with the corresponding copy in the the non-volitale memory.
- void gpNvm_Clear (UInt8 componentId, UInt8 tagId)

  Clears the content of the specified NVM-tag using the provided RAM location. The non-volatile memory specified by the combination of the component and tag will be updated to the default values (pDefaultValues). If there are no default values specified the content of the non-volatile memory will be set to 0xFF.
- void **Nvm_WriteByte** (UIntPtr address, UInt8 value, gpNvm_UpdateFrequency_t updateFrequency)
- UInt8 Nvm_ReadByte (UIntPtr address, gpNvm_UpdateFrequency_t updateFrequency)

  (defined(GP_NVM_DIVERSITY_TAG_IF))
- void gpNvm_SetLock (Bool lock)

  (defined(GP_NVM_DIVERSITY_TAG_IF))
- void gpNvm_CheckConsistency (void)

  Checks if NVM is consistent. The function checks if the NVM is consistent. If not, NVM is cleared.
- Bool gpNvm_CheckAccessible (void)

  Checks if NVM is accessible. The function checks if the NVM is accessible.
- void **Nvm_ReadBlock** (UIntPtr address, UInt16 lengthBlock, UInt8 ∗data, gpNvm_UpdateFrequency_t updateFrequency)
- void gpNvm_Refresh (void)

  (defined(GP_NVM_DIVERSITY_TAG_IF))

- gpNvm_Result_t **gpNvm_AcquireLutHandle** (gpNvm_LookupTable_Handle_t ∗pHandle, gpNvm_PoolId_t poolId, gpNvm_UpdateFrequency_t updateFrequencySpec, UInt8 tokenLength, UInt8 ∗pToken, Bool ∗freeAfterUse, gpNvm_KeyIndex_t maxNbrOfMatches)
- gpNvm_Result_t gpNvm_ReadNext (gpNvm_LookupTable_Handle_t handle, gpNvm_PoolId_t poolId, gpNvm_UpdateFrequency_t ∗pUpdateFrequency, UInt8 maxTokenLength, UInt8 ∗pTokenLength, UInt8 ∗pToken, UInt8 maxDataLength, UInt8 ∗pDataLength, UInt8 ∗pData)

  Retrieve the next data that matches the filter criteria from the NVM, this method requires the availability of a temporary lookup table You should always call gpNvm_FreeLookup when gpNvm_Result_DataAvailable or gpNvm_Result_Truncated is returned, and you are done working with the lookup table.
- gpNvm_Result_t gpNvm_ReadUnique (gpNvm_LookupTable_Handle_t handle, gpNvm_PoolId_t poolId, gpNvm_UpdateFrequency_t updateFrequencySpec, gpNvm_UpdateFrequency_t ∗pUpdateFrequency, UInt8 tokenLength, UInt8 ∗pToken, UInt8 maxDataLength, UInt8 ∗pDataLength, UInt8 ∗pData)

  method to retrieve data of an attribute which has a unique match of the specified filters in the temporary lookup table
- void gpNvm_FreeLookup (gpNvm_LookupTable_Handle_t handle)

  Method to free a lookuptable after use.
- gpNvm_Result_t gpNvm_Remove (gpNvm_PoolId_t poolId, gpNvm_UpdateFrequency_t updateFrequencySpec, UInt8 tokenLength, UInt8 ∗pToken)

  Remove an nvm entry.
- gpNvm_Result_t gpNvm_ErasePool (gpNvm_PoolId_t poolId)

  Erase the content of a specific pool.
- gpNvm_Result_t **gpNvm_GetNextTokenKey** (gpNvm_LookupTable_Handle_t handle, gpNvm_PoolId_t poolId, gpNvm_UpdateFrequency_t updateFrequencySpec, UInt8 tokenLength, UInt8 ∗pToken, UInt8 ∗pTokenKey)
- gpNvm_Result_t gpNvm_ResetIterator (gpNvm_LookupTable_Handle_t handle)

  Reset the iteration sequence of ReadNext to the first item.
- void gpNvm_cbFailedCheckConsistency (void)

  Callback triggered if consistency check failed.
- void gpNvm_SetBackgroundDefragmentationMode (Bool enable)

  Controls the automatic background defragmentation. To allow multiple usage a counter mechanism in implemented. The auto defragmentation is enabled by default.
- void gpNvm_RegisterSection (UInt8 componentId, const ROM gpNvm_Tag_t ∗tags, UInt8 nbrOfTags, gpNvm_cbCheckConsistency_t cbCheckConsistency)

  Registers a component to the NVM software block.
- gpNvm_Result_t **gpNvm_GetNvmType** (UInt8 ∗pNvmType)

- #define gpNvm_UpdateFrequencyHigh 0

  Updated more often, typically used for attributes that can get updated during normal user usage.
- #define gpNvm_UpdateFrequencyLow 1

  Updated in rare condition.
- #define gpNvm_UpdateFrequencyInitOnly 2

  Updated only during initialisation phase, typically used for attributes which are only set once.
- #define gpNvm_UpdateFrequencyReadOnly 3

  Set during configuration, read-only information.
- #define gpNvm_UpdateFrequencyIgnore 4

  Wildcard value used with read queries.
- #define **gpNvm_NrOfUpdateFrequencyTypes** 5
- typedef UInt8 gpNvm_UpdateFrequency_t

The gpNvm_UpdateFrequency_t type defines indicating the frequency a tag will require a new backup into NVM.

- #define **gpNvm_Result_DataAvailable** 0
- #define **gpNvm_Result_NoDataAvailable** 1
- #define **gpNvm_Result_NoLookUpTable** 2
- #define **gpNvm_Result_NoUniqueMaskMatch** 4
- #define **gpNvm_Result_Truncated** 5
- #define **gpNvm_Result_Error** 0xFF
- typedef UInt8 **gpNvm_Result_t**

- #define gpNvm_PoolId_Tag 0

  The pool used by the tag API.
- #define gpNvm_PoolId_Application1 1

  Application specified pool.
- #define gpNvm_PoolId_Application2 2

  Application specified pool.
- #define gpNvm_PoolId_Application3 3

  Application specified pool.
- #define gpNvm_PoolId_Application4 4

  Application specified pool.
- #define gpNvm_PoolId_AllPoolIds 0xFE

  Used to iterate all pools.
- typedef UInt8 gpNvm_PoolId_t

  TBD:Namespace prefix or physical pool id.

4.1.1 Macro Definition Documentation

GP_NVM_MAX_PAYLOADLENGTH

```
#define GP_NVM_MAX_PAYLOADLENGTH 241
```
GP_NVM_MAX_PAYLOADLENGTH

GP_NVM_MAX_TOKENLENGTH

```
#define GP_NVM_MAX_TOKENLENGTH 1
```
GP_NVM_MAX_TOKENLENGTH

GP_NVM_SECTIONID_TAG

```
#define GP_NVM_SECTIONID_TAG 16
```
GP_NVM_SECTIONID_TAG

gpNvm_AllComponents

```
#define gpNvm_AllComponents 0xEE
```
gpNvm_AllComponents

gpNvm_AllTags

```
#define gpNvm_AllTags 0xEE
```

gpNvm_AllTags

4.1.2 Typedef Documentation

gpNvm_cbCheckConsistency_t

```
typedef Bool(* gpNvm_cbCheckConsistency_t) (void)
```

to function gpNvm_cbCheckConsistency_t

Returns

status

gpNvm_cbUpdate_t

```
typedef Bool(* gpNvm_cbUpdate_t) (void)
```

to function gpNvm_cbUpdate_t

Returns

status

4.1.3 Function Documentation

gpNvm_Backup()

```
void gpNvm_Backup (
            UInt8 componentId,
            UInt8 tagId,
            UInt8 * pRamLocation )
```

Parameters

componentId	The unique idenditfier nvm-section of the section to be backuped, gpNvm_AllComponents can be used as wildcard.
tagId	The sequence number of the tag to be backuped
pRamLocation	Pointer to the data that should be backuped. When this pointer is NULL, the internal stored pointer is used.

gpNvm_CheckAccessible()

```
Bool gpNvm_CheckAccessible (
            void )
```

Returns

result Boolean that retunrns the result of the check.

gpNvm_Clear()

```
void gpNvm_Clear (
            UInt8 componentId,
            UInt8 tagId )
```

Parameters

componentId	The unique identifier nvm-section of the section to be cleared, gpNvm_AllComponents can be used as wildcard.
tagId	The sequence number of the tag to be cleard, gpNvm_AllTags can be used as wildcard.

gpNvm_Dump()

```
void gpNvm_Dump (
            UInt8 componentId,
            UInt8 tagId )
```

Parameters

componentId	The unique idenditfier nvm-section of the section to be restored, gpNvm_AllComponents can be used as wildcard.
tagId	The sequence number of the tag to be restored, gpNvm_AllTags can be used as wildcard.

gpNvm_ErasePool()

```
gpNvm_Result_t gpNvm_ErasePool (
            gpNvm_PoolId_t poolId )
```

Parameters

poolId	The pool ID to be erased. Use gpNvm_PoolId_AllPoolIds to erase all pools.

Returns

result gpNvm_Result_DataAvailable gpNvm_Result_Error

gpNvm_FreeLookup()

```
void gpNvm_FreeLookup (
            gpNvm_LookupTable_Handle_t handle )
```

Parameters

handle	lookuptable handle

gpNvm_ReadNext()

```
gpNvm_Result_t gpNvm_ReadNext (
            gpNvm_LookupTable_Handle_t handle,
            gpNvm_PoolId_t poolId,
            gpNvm_UpdateFrequency_t * pUpdateFrequency,
            UInt8 maxTokenLength,
            UInt8 * pTokenLength,
            UInt8 * pToken,
            UInt8 maxDataLength,
            UInt8 * pDataLength,
            UInt8 * pData )
```

Parameters

handle	lookuptable handle
poolId	The pool ID to be used
pUpdateFrequency	used to return the updateFrequency of the read data typically used by caller when lookup table is created with gpNvm_BuildLookupOverAllUpdateFrequencies; will be ignored when specifying NULL
maxTokenLength	maximum length in bytes of pToken
pTokenLength	filled in number of bytes in pToken typically used by caller when lookup table is created with tokenlength 0; will be ignored when specifying 0
pToken	pointer to buffer to return token information typically used by caller when lookup table is created with gpNvm_BuildLookupOverAllUpdateFrequencies; will be ignored when specifying NULL
maxDataLength	maximum length in bytes of pData
pDataLength	number of bytes in pData
pData	pointer to buffer to return data

Returns

result gpNvm_Result_Truncated in case there were more entries than requested

gpNvm_ReadUnique()

```
gpNvm_Result_t gpNvm_ReadUnique (
            gpNvm_LookupTable_Handle_t handle,
            gpNvm_PoolId_t poolId,
            gpNvm_UpdateFrequency_t updateFrequencySpec,
            gpNvm_UpdateFrequency_t * pUpdateFrequency,
            UInt8 tokenLength,
            UInt8 * pToken,
            UInt8 maxDataLength,
            UInt8 * pDataLength,
            UInt8 * pData )
```

Parameters

handle	lookuptable handle
poolId	The pool ID to be used
updateFrequencySpec	specify the updateFrequency of the read data, gpNvm_UpdateFrequencyIgnore can be used as wildcard
pUpdateFrequency	returns the update frequency of the attribute
tokenLength	length in bytes of pToken
pToken	array forming the unique reference tot the attribute within the updateFrequency scope - will result in unique match
maxDataLength	maximum length in bytes of pData
pDataLength	number of bytes in pData
pData	pointer to buffer to return data

Returns

result gpNvm_Result_DataAvailable gpNvm_Result_NoDataAvailable gpNvm_Result_NoLookUpTable gpNvm_Result_NoUniqueMaskMatch

gpNvm_Remove()

```
gpNvm_Result_t gpNvm_Remove (
            gpNvm_PoolId_t poolId,
            gpNvm_UpdateFrequency_t updateFrequencySpec,
            UInt8 tokenLength,
            UInt8 * pToken )
```

Parameters

poolId	The pool ID to be used
updateFrequencySpec	specify the updateFrequency of the read data, gpNvm_UpdateFrequencyIgnore can be used as wildcard
tokenLength	length in bytes of pToken
pToken	array forming the unique reference tot the attribute within the updateFrequency scope - will result in unique match

Returns

result gpNvm_Result_DataAvailable gpNvm_Result_Error

gpNvm_ReserveRemoteNvmSpace()

```
UInt16 gpNvm_ReserveRemoteNvmSpace (
            UInt16 requiredSize )
```

Parameters

requiredSize	Size of the NVM to be used remotely.

Returns

offset Offset within local NVM where remote NVM may store sections

gpNvm_ResetIterator()

```
gpNvm_Result_t gpNvm_ResetIterator (
            gpNvm_LookupTable_Handle_t handle )
```

Parameters

handle	lookuptable handle

Returns

result gpNvm_Result_DataAvailable gpNvm_Result_Error

gpNvm_Restore()

```
void gpNvm_Restore (
            UInt8 componentId,
            UInt8 tagId,
            UInt8 * pRamLocation )
```

Parameters

componentId	The unique idenditfier nvm-section of the section to be restored, gpNvm_AllComponents can be used as wildcard.
tagId	The sequence number of the tag to be restored.
pRamLocation	Pointer to the data that should be restored. When this pointer is NULL, the internal stored pointer is used.

gpNvm_SetBackgroundDefragmentationMode()

```
void gpNvm_SetBackgroundDefragmentationMode (
            Bool enable )
```

Parameters

enable	false = disable defragmentation (counter is incremented), true = enable defragmentation (counter is decremented). The number of calls with true should match the number with calls with false to enable the background defragmentation.

gpNvm_SetLock()

```
void gpNvm_SetLock (
            Bool lock )
```

Locks or unlocks the possibility to write or clear read-only elements. Default the NVM area will be unable to write or clear read-only elements (tags with update frequency gpNvm_UpdateFrequencyReadOnly). One can 'unlock' the area by calling this function with false. Re-lock the area after the needed manipulations by calling this function with true.

Parameters

lock	Lock (true) or Unlock (false) the NVM area.

4.2 gpNvm_ElemIf.h File Reference

Data Structures

- struct gpNvm_ElementDefine_t
- struct gpNvm_LookUpTableHeader_t
- struct gpNvm_IdentifiableTag_t

Macros

- #define GP_NVM_MAX_NUM_BASE_LUT_ELEMENTS 255

  GP_NVM_MAX_NUM_BASE_LUT_ELEMENTS is a macro defining the limit for the number of attributes inherited from the standard application.
- #define GP_NVM_MAX_NUM_LUT_ELEMENTS 15

  GP_NVM_MAX_NUM_LUT_ELEMENTS is a macro defining the limit for the number of attributes defined in NVM based LookUp Table.
- #define GP_NVM_NO_LUT_INDEX 0xFFFF

  GP_NVM_NO_LUT_INDEX is a macro defining the value used in the list link field of the last LUT in the list.
- #define NVM_MAX_TAG_SIZE 239

  NVM_MAX_TAG_SIZE is a macro defining the maximal size in bytes of object located in the NVM element.
- #define GP_NVM_VERSION_CRC_UNIQUE_TAG_ID 0

GP_NVM_VERSION_CRC_UNIQUE_TAG_ID is a macro defining the tag ID of the inherited NVM CRC attribute. Must allways be 0.

- #define GP_NVM_TYPE_UNIQUE_TAG_ID 0xEF

  Wildcard used to access Nvm type.
- #define GP_NVM_REMOTE_RW_TAG_ID_BASE 128

  GP_NVM_REMOTE_RW_TAG_ID_BASE is a macro defining the tag ID base used for elements storing remote NVM. This is the value that should be used for all of the elements registered for remote NVM.

Typedefs

- typedef Bool(∗ gpNvm_cbDefaultValueInitializer_t) (const ROM void ∗pTag, UInt8 ∗pBuffer)

  The gpNvm_cbDefaultValueInitializer_t type describes reference to a function that initializes attribute with default values.
- typedef Bool(∗ gpNvm_cbConsistencyChecker_t) (const ROM gpNvm_IdentifiableTag_t ∗pTag)

  The gpNvm_cbConsistencyChecker_t type describes reference to a function that provides additional element specific check-consistency routine.

Functions

- gpNvm_Result_t gpNvm_GetNvmType (UInt8 ∗pNvmType)

  Get the Nvm Type describing the Nvm building blocks used.
- void gpNvm_Config (const ROM gpNvm_LookUpTableHeader_t ∗baseTableHeader, const ROM gpNvm_ElementDefine_t ∗baseTableElements, const ROM gpNvm_VersionCrc_t ∗inheritedNvmVersion

  Configures NVM for base tables other than gpNvm_elementBaseLUTHeader and gpNvm_elementBaseLUTElements.
- void gpNvm_RegisterElement (const ROM gpNvm_IdentifiableTag_t ∗pTag)

  Registers an element to the NVM software block.
- void gpNvm_RegisterElements (const ROM gpNvm_IdentifiableTag_t ∗pTag, UInt8 nbrOfTags)

  Registers multiple elements to the NVM software block.

4.2.1 Macro Definition Documentation

GP_NVM_TYPE_UNIQUE_TAG_ID

```
#define GP_NVM_TYPE_UNIQUE_TAG_ID 0xEF
```

gpNvm_VersionKey

4.2.2 Typedef Documentation

gpNvm_cbConsistencyChecker_t

```
gpNvm_cbConsistencyChecker_t
```

Parameters

pTag	points to the description of the element to be checked

Returns

true if the check passed and false otherwise

gpNvm_cbDefaultValueInitializer_t

```
gpNvm_cbDefaultValueInitializer_t
```

Parameters

pTag	points to the description of the element to be initialized. If pTag indicates non NULL pRamLocation default values will be stored there.
pBuffer	points to buffer that will be used (if not NULL) to store default values.

Returns

true if the initialization was successful and false otherwise

4.2.3 Function Documentation

gpNvm_GetNvmType()

```
gpNvm_Result_t gpNvm_GetNvmType (
            UInt8 * pNvmType )
```

Parameters

pNvmType	Pointer to UInt8 to store NvmType value.

Returns

result gpNvm_Result_DataAvailable gpNvm_Result_NoDataAvailable