Document Title	Specification of NVRAM Manager
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	033

Document Status	published
Part of AUTOSAR Standard	Classic Platform
Part of Standard Release	R21-11

Document Change History			
Date	**Release**	**Changed by**	**Description**
2021-11-25	R21-11	AUTOSAR Release Management	• Changes related to the concept 691 MemoryStackRework • Clarification regarding validation in NvM_WriteBlock • Migration from doc to latex
2020-11-30	R20-11	AUTOSAR Release Management	• InitBlockCallback and ROM block are mutually exclusive • Removal for DET error NVM_E_PARAM_BLOCK_TYPE • NvM partitioning for multi-core
2019-11-28	R19-11	AUTOSAR Release Management	• Changes related to NVM_E_WRITE_PROTECTED • Port Prototypes are generated for block only if needed • Changed Document Status from Final to published
2018-10-31	4.4.0	AUTOSAR Release Management	• Removed NvM_GetActiveService API • Remove EcuMfixed completely • Changed single and multi block callbacks • minor corrections / clarifications / editorial changes; For details please refer to the ChangeDocumentation

2017-12-08	4.3.1	AUTOSAR Release Management	• Correction for write protection and erase requests for NvMWriteBlockOnce blocks • Clarification regarding implicit recovery of dataset blocks • minor corrections / clarifications / editorial changes; For details please refer to the ChangeDocumentation
2016-11-30	4.3.0	AUTOSAR Release Management	• Added NvM_FirstInitAll and NvM_GetActiveService functionalities • NvM_SetRamBlockStatus works also for explicit synchronization blocks • The interaction between NvM and BswM is clarified. • Other small clarifications and updates.
2015-07-31	4.2.2	AUTOSAR Release Management	• Clarified behavior related to restoring default data for blocks and for handling of MEMIF_BLOCK_INVALID job result • Added additional information related to the block states in chapter 7.2.2.14 and related subchapters • Updated NvM_Init and NvM_ValidateAll function prototypes • Debugging support marked as obsolete
2014-10-31	4.2.1	AUTOSAR Release Management	• Detailed pass/fail conditions for production errors • Added the NvM_ValidateAll functionality • Updated return values for Init and SingleBlock callbacks • Other small clarifications
2014-03-31	4.1.3	AUTOSAR Release Management	• Removed job postpone in case of explicit synchronization failed after configured number of retries • Updated Service Interfaces tables • Renamed configuration parameter NvMRamBlockHeaderInclude to NvMBlockHeaderInclude • Editorial changes

2013-10-31	4.1.2	AUTOSAR Release Management	• Added NvMRamBlockHeaderInclude and NvMMainFunctionPeriod configuration parameters • Corrected bugs for NvMWriteVerificationDataSize and NvMNvramBlockIdentifier parameters • Other small clarifications in requirement • Editorial changes • Removed chapter(s) on change documentation
2013-03-15	4.1.1	AUTOSAR Release Management	• Added NvM_ReadPRAMBlock, NvM_WritePRAMBlock and NvM_ RestorePRAMBlockDefaults APIs • Production Errors and Extended Production Errors classification • Clarifications for explicit synchronization mechanism • Modeling of Services: introduction of formal descriptions of service interfaces • Changes regarding NvM_ CancelJobs API, NvmSetRamBlockStatus API, Init callback, handling of redundant blocks,queue sizes and usage of MemoryMapping • Reworked according to the new SWS_BSWGeneral
2011-12-22	4.0.3	AUTOSAR Release Management	• Added NvM_CancelJobs behaviour • Added NvM and BswM interaction • Added NvM_SetBlockLockStatus API functional description • Corrected inconsistency between C-interface and port interface • Updated Include structure • Updated configuration parameters description and range

2010-09-30	3.1.5	AUTOSAR Release Management	• Behavior specified to prevent possible loss of data during shutdown • References to DEM for production errors, new config container NvmDemEventParameterRefs • NvMMaxNoOfWriteRetries renamed to NvMMaxNumOfWriteRetries • Note in chapter 7.1.4.5 completed • Null pointer handling changed • Chapter "Version check" updated • New DET error NVM_E_PARAM_POINTER • Chapter 10 updated, NvMMainFunctionCycleTime moved, NvMSelectBlockForWriteAll added, some ranges corrected • Behavior specified when NVRAM block ID 1 shall be written • Chapter 12 updated • Handling of single-block callbacks during asynchronous multi-block specified. • Some minor changes, typos corrected
2010-02-02	3.1.4	AUTOSAR Release Management	• The following features had impact on this document: • Debugging concept • Error handler concept • Memory related concepts • The following major features were necessary to implement these concepts: • Static Block Id Check • Write Verification • Read Retry • buffered read/write-operations • Legal disclaimer revised

2008-08-13	3.1.1	AUTOSAR Release Management	• Technical Office SWS Improvements are incorporated. • Requirement IDs for configuration parameters (chapter 10) added. • Management of the RAM block state specified more precisely. • The NVRAM Manager doesn't support non-sequential NVRAM block IDs any longer. • Document meta information extended • Small layout adaptations made
2007-12-21	3.0.1	AUTOSAR Release Management	• Legal disclaimer revised
2007-01-24	2.1.15	AUTOSAR Release Management	• AUTOSAR service description added in chapter 11 • Reentrancy of callback functions specified • Details regarding memory hardware abstraction addressing scheme added • Legal disclaimer revised • "Advice for users" revised • "Revision Information" added
2006-05-16	2.0	AUTOSAR Release Management	• Document structure adapted to common Release 2.0 SWS Template. • Major changes in chapter 10 • Structure of document changed partly
2005-05-31	1.0	AUTOSAR Release Management	• Initial release

Disclaimer

This work (specification and/or software implementation) and the material contained in it, as released by AUTOSAR, is for the purpose of information only. AUTOSAR and the companies that have contributed to it shall not be liable for any use of the work.

The material contained in this work is protected by copyright and other types of intellectual property rights. The commercial exploitation of the material contained in this work requires a license to such intellectual property rights.

This work may be utilized or reproduced without any modification, in any form or by any means, for informational purposes only. For any other purpose, no part of the work may be utilized or reproduced, in any form or by any means, without permission in writing from the publisher.

The work has been developed for automotive applications only. It has neither been developed, nor tested for non-automotive applications.

The word AUTOSAR and the AUTOSAR logo are registered trademarks.

Table of Contents

1 Introduction and functional overview 11

2 Acronyms and Abbreviations 13

3 Related documentation 14

3.1 Input documents & related standards and norms 14
3.2 Related specification 14

4 Constraints and assumptions 15

4.1 Limitations 15
4.2 Applicability to car domains 15
4.3 Conflicts 15

5 Dependencies to other modules 15

5.1 File structure 16
5.1.1 Header file structure 16
5.2 Memory abstraction modules 16
5.3 CRC module 16
5.4 Capability of the underlying drivers 16

6 Requirements Tracing 16

7 Functional specification 25

7.1 Basic architecture guidelines 25
7.1.1 Layer structure 25
7.1.2 Addressing scheme for the memory hardware abstraction . . 25
7.1.2.1 Examples 26
7.1.3 Basic storage objects 27
7.1.3.1 NV block 27
7.1.3.2 RAM block 28
7.1.3.3 ROM block 29
7.1.3.4 Administrative block 30
7.1.3.5 NV Block Header 30
7.1.4 Block management types 31
7.1.4.1 Block management types overview 31
7.1.4.2 NVRAM block structure 32
7.1.4.3 NVRAM block descriptor table 32
7.1.4.4 Native NVRAM block 33
7.1.4.5 Redundant NVRAM block 33
7.1.4.6 Dataset NVRAM block 35
7.1.4.7 NVRAM Manager API configuration classes 36
7.1.5 Scan order / priority scheme 39
7.2 General behavior 40
7.2.1 Functional requirements 40
7.2.2 Design notes 42

7.2.2.1 NVRAM manager startup . . . 42
7.2.2.2 NVRAM manager shutdown . . . 43
7.2.2.3 (Quasi) parallel write access to the NvM module . . . 43
7.2.2.4 NVRAM block consistency check . . . 43
7.2.2.5 Error recovery . . . 44
7.2.2.6 Recovery of a RAM block with ROM data . . . 44
7.2.2.7 Implicit recovery of a RAM block with ROM default data 44
7.2.2.8 Explicit recovery of a RAM block with ROM default data 45
7.2.2.9 Detection of an incomplete write operation to a NV block . . . 46
7.2.2.10 Termination of a single block request . . . 46
7.2.2.11 Termination of a multi block request . . . 46
7.2.2.12 General handling of asynchronous requests/ job processing . . . 47
7.2.2.13 NVRAM block write protection . . . 47
7.2.2.14 Validation and modification of RAM block data . . . 48
7.2.2.15 Communication and implicit synchronization between application and NVRAM manager . . . 52
7.2.2.16 Normal and extended runtime preparation of NVRAM blocks . . . 55
7.2.2.17 Communication and explicit synchronization between application and NVRAM manager . . . 56
7.2.2.18 Static Block ID Check . . . 60
7.2.2.19 Read Retry . . . 61
7.2.2.20 Write Verification . . . 61
7.2.2.21 Comparing NV data in NvM . . . 62
7.2.2.22 NvM and BswM interaction . . . 62
7.2.2.23 NvM behaviour in case of Block locked . . . 63
7.2.2.24 Block Compression . . . 64
7.2.2.25 Block Ciphering . . . 65
7.3 Error Classification . . . 67
7.3.1 Development Errors . . . 67
7.3.2 Runtime Errors . . . 73
7.3.3 Transient Faults . . . 74
7.3.4 Production Errors . . . 74
7.3.4.1 NVM_E_HARDWARE . . . 74
7.3.5 Extended Production Errors . . . 75
7.3.5.1 NVM_E_INTEGRITY_FAILED . . . 75
7.3.5.2 NVM_E_REQ_FAILED . . . 76
7.3.5.3 NVM_E_WRONG_BLOCK_ID . . . 77
7.3.5.4 NVM_E_VERIFY_FAILED . . . 77
7.3.5.5 NVM_E_LOSS_OF_REDUNDANCY . . . 78
8 API specification 78
8.1 Imported types . . . 78
8.2 Type definitions . . . 79

8.2.1 NvM_ConfigType . . . 79
8.2.2 NvM_MultiBlockRequestType . . . 79
8.3 Function definitions . . . 79
8.3.1 Synchronous requests . . . 79
8.3.1.1 NvM_Init . . . 79
8.3.1.2 NvM_SetDataIndex . . . 80
8.3.1.3 NvM_GetDataIndex . . . 81
8.3.1.4 NvM_SetBlockProtection . . . 82
8.3.1.5 NvM_GetErrorStatus . . . 83
8.3.1.6 NvM_GetVersionInfo . . . 83
8.3.1.7 NvM_SetRamBlockStatus . . . 84
8.3.1.8 NvM_SetBlockLockStatus . . . 86
8.3.1.9 NvM_CancelJobs . . . 87
8.3.2 Asynchronous single block requests . . . 88
8.3.2.1 NvM_ReadBlock . . . 88
8.3.2.2 NvM_WriteBlock . . . 91
8.3.2.3 NvM_RestoreBlockDefaults . . . 94
8.3.2.4 NvM_EraseNvBlock . . . 96
8.3.2.5 NvM_InvalidateNvBlock . . . 97
8.3.2.6 NvM_ReadPRAMBlock . . . 98
8.3.2.7 NvM_WritePRAMBlock . . . 101
8.3.2.8 NvM_RestorePRAMBlockDefaults . . . 104
8.3.3 Asynchronous multi block requests . . . 105
8.3.3.1 NvM_ReadAll . . . 105
8.3.3.2 NvM_WriteAll . . . 112
8.3.3.3 NvM_CancelWriteAll . . . 116
8.3.3.4 NvM_ValidateAll . . . 117
8.3.3.5 NvM_FirstInitAll . . . 118
8.3.3.6 Callback notifications . . . 122
8.4 Scheduled functions . . . 123
8.5 Expected interfaces . . . 125
8.5.1 Mandatory Interfaces . . . 125
8.5.2 Optional Interfaces . . . 125
8.5.3 Configurable interfaces . . . 126
8.6 API Overview . . . 130
8.7 Service Interfaces . . . 131
8.7.1 Client-Server-Interfaces . . . 131
8.7.1.1 NvM_Admin . . . 131
8.7.1.2 NvM_Mirror . . . 131
8.7.1.3 NvM_NotifyInitBlock . . . 132
8.7.1.4 NvM_NotifyJobFinished . . . 133
8.7.1.5 NvM_Service . . . 133
8.7.2 Implementation Data Types . . . 137
8.7.2.1 ImplementationDataType NvM_RequestResultType . 138
8.7.2.2 ImplementationDataType NvM_BlockIdType . . . 139
8.7.2.3 ImplementationDataType NvM_InitBlockRequestType 139

8.7.2.4 ImplementationDataType NvM_BlockRequestType . 139
8.7.3 Ports . . . 140
8.7.3.1 NvM_PAdmin_{Block} . . . 140
8.7.3.2 NvM_PM_{Block} . . . 141
8.7.3.3 NvM_PNIB_{Block} . . . 141
8.7.3.4 NvM_PNJF_{Block} . . . 142
8.7.3.5 NvM_PS_{Block} . . . 142

9 Sequence diagrams 143

9.1 Synchronous calls . . . 143
9.1.1 NvM_Init . . . 143
9.1.2 NvM_SetDataIndex . . . 144
9.1.3 NvM_GetDataIndex . . . 144
9.1.4 NvM_SetBlockProtection . . . 145
9.1.5 NvM_GetErrorStatus . . . 145
9.1.6 NvM_GetVersionInfo . . . 146
9.2 Asynchronous calls . . . 146
9.2.1 Asynchronous call with polling . . . 146
9.2.2 Asynchronous call with callback . . . 147
9.2.3 Cancellation of a Multi Block Request . . . 148
9.2.4 BswM Interraction . . . 149

10 Configuration specification 153

10.1 How to read this chapter . . . 153
10.2 Containers and configuration parameters . . . 154
10.2.1 NvM . . . 154
10.2.2 NvMCommon . . . 155
10.2.3 NvMBlockDescriptor . . . 163
10.2.4 NvMInitBlockCallback . . . 181
10.2.5 NvMSingleBlockCallback . . . 182
10.2.6 NvMTargetBlockReference . . . 183
10.2.7 NvMEaRef . . . 183
10.2.8 NvMFeeRef . . . 184
10.2.9 NvmDemEventParameterRefs . . . 184
10.2.10 NvMBlockCiphering . . . 187
10.3 Published Information . . . 190

A Not applicable requirements 190

1 Introduction and functional overview

This specification describes the functionality, API and the configuration of the AUTOSAR Basic Software module NVRAM Manager (NvM).

The NvM module shall provide services to ensure the data storage and maintenance of NV (non volatile) data according to their individual requirements in an automotive environment. The NvM module shall be able to administrate the NV data of an EEPROM and/or a FLASH EEPROM emulation device.

The NvM module shall provide the required synchronous/asynchronous services for the management and the maintenance of NV data (init/read/write/control).

The relationship between the different blocks can be visualized in the following picture:

-

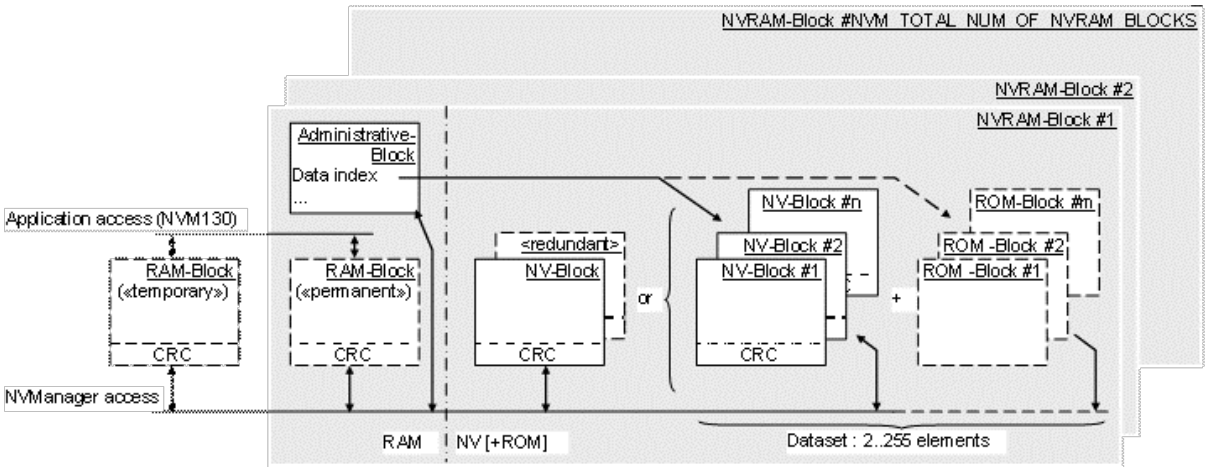

Figure 1.1: Memory Structure of Different Block Types

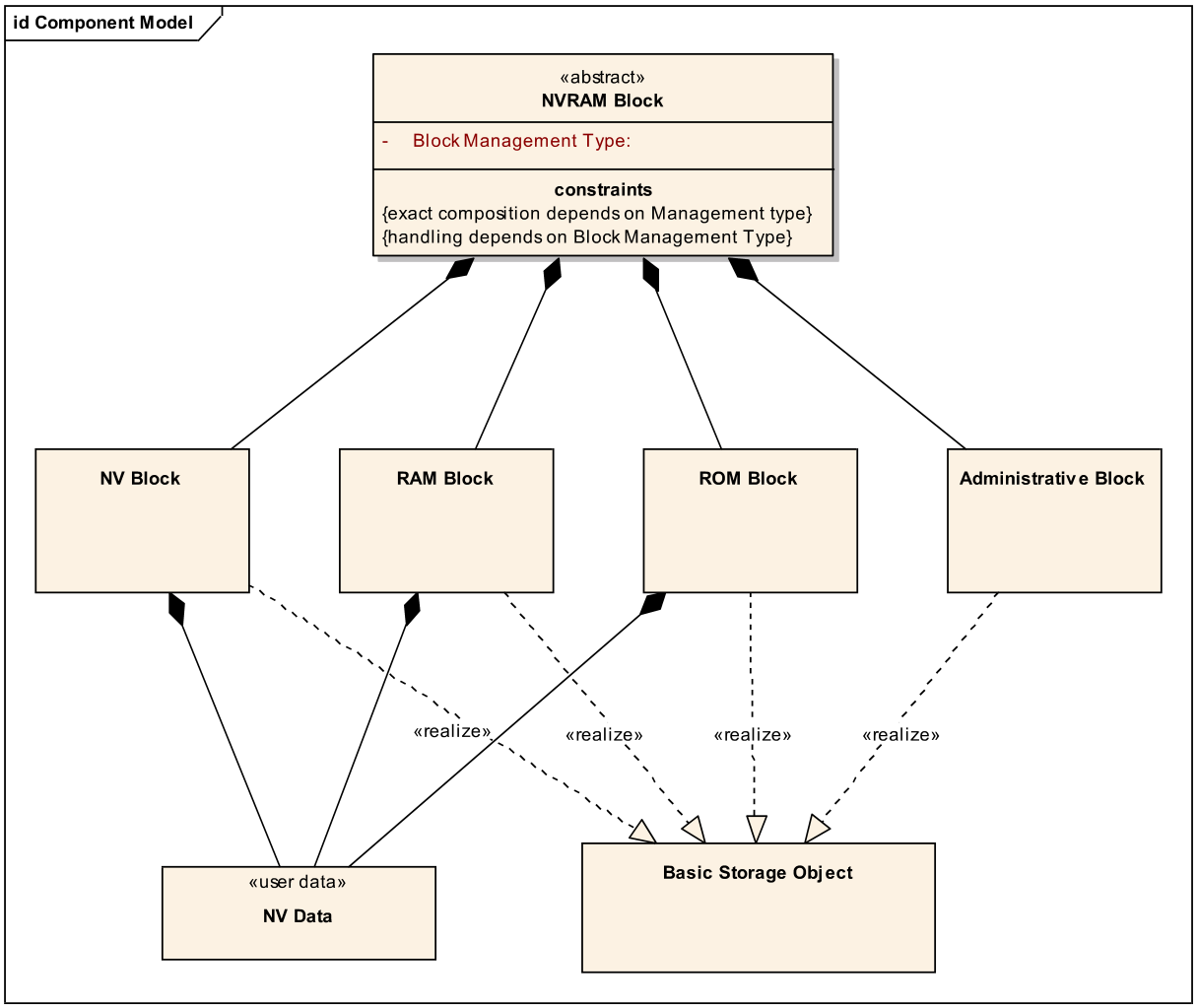

Figure 1.2: Logical Structure of Different Block Types

2 Acronyms and Abbreviations

The glossary below includes acronyms and abbreviations relevant to the NvM module that are not included in the [1, AUTOSAR glossary].

Abbreviation / Acronym:	Description:
• Basic Storage Object	• A "Basic Storage Object" is the smallest entity of a "NVRAM block". Several "Basic Storage Objects" can be used to build a NVRAM Block. A "Basic Storage Object" can reside in different memory locations (RAM/ROM/NV memory).
• NVRAM Block	• The "NVRAM Block" is the entire structure, which is needed to administrate and to store a block of NV data.
• NV data	• The data to be stored in Non-Volatile memory.
• Block Management Type	• Type of the NVRAM Block. It depends on the (configurable) individual composition of a NVRAM Block in chunks of different mandatory/optional Basic Storage Objects and the subsequent handling of this NVRAM block.
• RAM Block	• The "RAM Block" is a "Basic Storage Object". It represents the part of a "NVRAM Block" which resides in the RAM. • See [SWS_NvM_00126]
• ROM Block	• The "ROM Block" is a "Basic Storage Object". It represents the part of a "NVRAM Block" which resides in the ROM. The "ROM Block" is an optional part of a "NVRAM Block". [SWS_NvM_00020]
• NV Block	• The "NV Block" is a "Basic Storage Object". It represents the part of a "NVRAM Block" which resides in the NV memory. The "NV Block" is a mandatory part of a "NVRAM Block". [SWS_NvM_00125]
• NV Block Header	• Additional information included in the NV Block if the mechanism "Static Block ID" is enabled.
• Administrative Block	• The "Administrative Block" is a "Basic Storage Object". It resides in RAM. The "Administrative Block" is a mandatory part of a "NVRAM Block". [SWS_NvM_00135]
• DET	• Default Error Tracer - module to which development errors are reported.
• DEM	• Diagnostic Event Manager - module to which production relevant errors are reported
• NV	• Non volatile
• FEE	• Flash EEPROM Emulation
• EA	• EEPROM Abstraction
• FCFS	• First come first served

3 Related documentation

3.1 Input documents & related standards and norms

[1] Glossary
AUTOSAR_TR_Glossary

[2] General Specification of Basic Software Modules
AUTOSAR_SWS_BSWGeneral

[3] Layered Software Architecture
AUTOSAR_EXP_LayeredSoftwareArchitecture

[4] Specification of EEPROM Abstraction
AUTOSAR_SWS_EEPROMAbstraction

[5] Specification of Flash EEPROM Emulation
AUTOSAR_SWS_FlashEEPROMEmulation

[6] Specification of Memory Abstraction Interface
AUTOSAR_SWS_MemoryAbstractionInterface

[7] Specification of CRC Routines
AUTOSAR_SWS_CRCLibrary

[8] Specification of EEPROM Driver
AUTOSAR_SWS_EEPROMDriver

[9] Specification of Flash Driver
AUTOSAR_SWS_FlashDriver

[10] Requirements on I/O Hardware Abstraction
AUTOSAR_SRS_IOHWAbstraction

[11] Requirements on Memory Services
AUTOSAR_SRS_MemoryServices

[12] General Requirements on Basic Software Modules
AUTOSAR_SRS_BSWGeneral

[13] Requirements on Software Component Template
AUTOSAR_RS_SoftwareComponentTemplate

[14] Specification of RTE Software
AUTOSAR_SWS_RTE

3.2 Related specification

AUTOSAR provides a General Specification on Basic Software modules [2, SWS BSW General], which is also valid for NVRAMManager.

Thus, the specification SWS BSW General shall be considered as additional and required specification for NVRAMManager.

4 Constraints and assumptions

4.1 Limitations

Limitations are given mainly by the finite number of "Block Management Types" and their individual treatment of NV data. These limits can be reduced by an enhanced user defined management information, which can be stored as a structured part of the real NV data. In this case the user defined management information has to be interpreted and handled by the application at least.

4.2 Applicability to car domains

No restrictions.

4.3 Conflicts

The NvM can be configured to use functionality from other modules or integrator code. Examples include the en/decryption of block data using Csm or the compression of block data. It is the responsibility of the integrator to ensure that:

- the required **functionality is available** at the time NvM uses it (e.g. the called Csm is already initialized [or not yet de-initialized]; needed main functions in called modules are executed; ...)
- the required **time is available** (e.g. cryptographic algorithms may need some time and therefore the read/write functionality of the NvM may take much longer for blocks which need an en/decryption)

5 Dependencies to other modules

This section describes the relations to other modules within the basic software.

5.1 File structure

5.1.1 Header file structure

The include file structure shall be as follows:

[SWS_NvM_00554] ⌈NvM module shall include NvM.h, Dem.h, MemIf.h.⌋ *()*

[SWS_NvM_00691] ⌈Only NvM.h shall be included by the upper layer.⌋ *()*

5.2 Memory abstraction modules

The memory abstraction modules abstract the NvM module from the subordinated drivers which are hardware dependent [ref. to doc. [3]]. The memory abstraction modules provide a runtime translation of each block access initiated by the NvM module to select the corresponding driver functions which are unique for all configured EEPROM or FLASH storage devices. The memory abstraction module is chosen via the NVRAM block device ID which is configured for each NVRAM block. NvM access the memory abstraction modules through memory abstraction interface module, Mem If. [ref. to doc. [4], [5], [6]]

5.3 CRC module

The NvM module uses CRC generation routines (8/16/32 bit) to check and to generate CRC for NVRAM blocks as a configurable option. The CRC routines have to be provided externally [ref. to ch. 8.5.2].[ref. to doc. [7]]

5.4 Capability of the underlying drivers

A set of underlying driver functions has to be provided for every configured NVRAM device as, for example, internal or external EEPROM or FLASH devices. The unique driver functions inside each set of driver functions are selected during runtime via a memory hardware abstraction module (see chapter 5.2). A set of driver functions has to include all the needed functions to write to, to read from or to maintain (e.g. erase) a configured NVRAM device.[ref. to doc. [8], [9]]

6 Requirements Tracing

The following tables reference the requirements specified in [10], [11], [12] and links to the fulfillment of these. Please note that if column “Satisfied by” is empty for a specific requirement this means that this requirement is not fulfilled by this document.

Requirement	Description	Satisfied by
[SRS_BSW_00005]	Modules of the μC Abstraction Layer (MCAL) may not have hard coded horizontal interfaces	[SWS_NvM_00744]
[SRS_BSW_00006]	The source code of software modules above the μC Abstraction Layer (MCAL) shall not be processor and compiler dependent.	[SWS_NvM_00744]
[SRS_BSW_00007]	All Basic SW Modules written in C language shall conform to the MISRA C 2012 Standard.	[SWS_NvM_00744]
[SRS_BSW_00009]	All Basic SW Modules shall be documented according to a common standard.	[SWS_NvM_00744]
[SRS_BSW_00010]	The memory consumption of all Basic SW Modules shall be documented for a defined configuration for all supported platforms.	[SWS_NvM_00744]
[SRS_BSW_00101]	The Basic Software Module shall be able to initialize variables and hardware in a separate initialization function	[SWS_NvM_00399] [SWS_NvM_00400]
[SRS_BSW_00160]	Configuration files of AUTOSAR Basic SW module shall be readable for human beings	[SWS_NvM_00744]
[SRS_BSW_00161]	The AUTOSAR Basic Software shall provide a microcontroller abstraction layer which provides a standardized interface to higher software layers	[SWS_NvM_00744]
[SRS_BSW_00162]	The AUTOSAR Basic Software shall provide a hardware abstraction layer	[SWS_NvM_00744]
[SRS_BSW_00164]	The Implementation of interrupt service routines shall be done by the Operating System, complex drivers or modules	[SWS_NvM_00744]
[SRS_BSW_00168]	SW components shall be tested by a function defined in a common API in the Basis-SW	[SWS_NvM_00744]
[SRS_BSW_00170]	The AUTOSAR SW Components shall provide information about their dependency from faults, signal qualities, driver demands	[SWS_NvM_00744]
[SRS_BSW_00172]	The scheduling strategy that is built inside the Basic Software Modules shall be compatible with the strategy used in the system	[SWS_NvM_00464]
[SRS_BSW_00302]	All AUTOSAR Basic Software Modules shall only export information needed by other modules	[SWS_NvM_00744]

Requirement	Description	Satisfied by
[SRS_BSW_00304]	All AUTOSAR Basic Software Modules shall use only AUTOSAR data types instead of native C data types	[SWS_NvM_00744]
[SRS_BSW_00306]	AUTOSAR Basic Software Modules shall be compiler and platform independent	[SWS_NvM_00744]
[SRS_BSW_00307]	Global variables naming convention	[SWS_NvM_00744]
[SRS_BSW_00308]	AUTOSAR Basic Software Modules shall not define global data in their header files, but in the C file	[SWS_NvM_00744]
[SRS_BSW_00309]	All AUTOSAR Basic Software Modules shall indicate all global data with read-only purposes by explicitly assigning the const keyword	[SWS_NvM_00744]
[SRS_BSW_00312]	Shared code shall be reentrant	[SWS_NvM_00744]
[SRS_BSW_00314]	All internal driver modules shall separate the interrupt frame definition from the service routine	[SWS_NvM_00744]
[SRS_BSW_00321]	The version numbers of AUTOSAR Basic Software Modules shall be enumerated according specific rules	[SWS_NvM_00744]
[SRS_BSW_00323]	All AUTOSAR Basic Software Modules shall check passed API parameters for validity	[SWS_NvM_00027]
[SRS_BSW_00325]	The runtime of interrupt service routines and functions that are running in interrupt context shall be kept short	[SWS_NvM_00744]
[SRS_BSW_00327]	Error values naming convention	[SWS_NvM_00027] [SWS_NvM_91004]
[SRS_BSW_00328]	All AUTOSAR Basic Software Modules shall avoid the duplication of code	[SWS_NvM_00744]
[SRS_BSW_00330]	It shall be allowed to use macros instead of functions where source code is used and runtime is critical	[SWS_NvM_00744]
[SRS_BSW_00331]	All Basic Software Modules shall strictly separate error and status information	[SWS_NvM_00027] [SWS_NvM_91004]
[SRS_BSW_00333]	For each callback function it shall be specified if it is called from interrupt context or not	[SWS_NvM_00467] [SWS_NvM_00468] [SWS_NvM_00469]
[SRS_BSW_00334]	All Basic Software Modules shall provide an XML file that contains the meta data	[SWS_NvM_00744]
[SRS_BSW_00335]	Status values naming convention	[SWS_NvM_00744]

Requirement	Description	Satisfied by
[SRS_BSW_00336]	Basic SW module shall be able to shutdown	[SWS_NvM_00744]
[SRS_BSW_00337]	Classification of development errors	[SWS_NvM_91004]
[SRS_BSW_00341]	Module documentation shall contains all needed informations	[SWS_NvM_00744]
[SRS_BSW_00342]	It shall be possible to create an AUTOSAR ECU out of modules provided as source code and modules provided as object code, even mixed	[SWS_NvM_00744]
[SRS_BSW_00343]	The unit of time for specification and configuration of Basic SW modules shall be preferably in physical time unit	[SWS_NvM_00744]
[SRS_BSW_00344]	BSW Modules shall support link-time configuration	[SWS_NvM_00744]
[SRS_BSW_00347]	A Naming seperation of different instances of BSW drivers shall be in place	[SWS_NvM_00744]
[SRS_BSW_00348]	All AUTOSAR standard types and constants shall be placed and organized in a standard type header file	[SWS_NvM_00744]
[SRS_BSW_00353]	All integer type definitions of target and compiler specific scope shall be placed and organized in a single type header	[SWS_NvM_00744]
[SRS_BSW_00360]	AUTOSAR Basic Software Modules callback functions are allowed to have parameters	[SWS_NvM_00467] [SWS_NvM_00468] [SWS_NvM_00469]
[SRS_BSW_00361]	All mappings of not standardized keywords of compiler specific scope shall be placed and organized in a compiler specific type and keyword header	[SWS_NvM_00744]
[SRS_BSW_00371]	No description	[SWS_NvM_00744]
[SRS_BSW_00373]	The main processing function of each AUTOSAR Basic Software Module shall be named according the defined convention	[SWS_NvM_00464]
[SRS_BSW_00375]	Basic Software Modules shall report wake-up reasons	[SWS_NvM_00744]
[SRS_BSW_00378]	AUTOSAR shall provide a boolean type	[SWS_NvM_00744]
[SRS_BSW_00383]	The Basic Software Module specifications shall specify which other configuration files from other modules they use at least in the description	[SWS_NvM_00465] [SWS_NvM_00466]

Requirement	Description	Satisfied by
[SRS_BSW_00384]	The Basic Software Module specifications shall specify at least in the description which other modules they require	[SWS_NvM_00465] [SWS_NvM_00466]
[SRS_BSW_00385]	List possible error notifications	[SWS_NvM_00027] [SWS_NvM_91004]
[SRS_BSW_00386]	The BSW shall specify the configuration for detecting an error	[SWS_NvM_00027] [SWS_NvM_91004]
[SRS_BSW_00398]	The link-time configuration is achieved on object code basis in the stage after compiling and before linking	[SWS_NvM_00744]
[SRS_BSW_00399]	Parameter-sets shall be located in a separate segment and shall be loaded after the code	[SWS_NvM_00744]
[SRS_BSW_00400]	Parameter shall be selected from multiple sets of parameters after code has been loaded and started	[SWS_NvM_00744]
[SRS_BSW_00404]	BSW Modules shall support post-build configuration	[SWS_NvM_00744]
[SRS_BSW_00405]	BSW Modules shall support multiple configuration sets	[SWS_NvM_00744]
[SRS_BSW_00406]	A static status variable denoting if a BSW module is initialized shall be initialized with value 0 before any APIs of the BSW module is called	[SWS_NvM_00027] [SWS_NvM_00399] [SWS_NvM_00400] [SWS_NvM_91004]
[SRS_BSW_00412]	No description	[SWS_NvM_00744]
[SRS_BSW_00414]	Init functions shall have a pointer to a configuration structure as single parameter	[SWS_NvM_00447]
[SRS_BSW_00415]	Interfaces which are provided exclusively for one module shall be separated into a dedicated header file	[SWS_NvM_00744]
[SRS_BSW_00416]	The sequence of modules to be initialized shall be configurable	[SWS_NvM_00744]
[SRS_BSW_00417]	Software which is not part of the SW-C shall report error events only after the DEM is fully operational.	[SWS_NvM_00744]
[SRS_BSW_00422]	Pre-de-bouncing of error status information is done within the DEM	[SWS_NvM_00744]
[SRS_BSW_00423]	BSW modules with AUTOSAR interfaces shall be describable with the means of the SW-C Template	[SWS_NvM_00744]
[SRS_BSW_00425]	The BSW module description template shall provide means to model the defined trigger conditions of schedulable objects	[SWS_NvM_00464]

Requirement	Description	Satisfied by
[SRS_BSW_00426]	BSW Modules shall ensure data consistency of data which is shared between BSW modules	[SWS_NvM_00744]
[SRS_BSW_00427]	ISR functions shall be defined and documented in the BSW module description template	[SWS_NvM_00744]
[SRS_BSW_00429]	Access to OS is restricted	[SWS_NvM_00332]
[SRS_BSW_00432]	Modules should have separate main processing functions for read/receive and write/transmit data path	[SWS_NvM_00744]
[SRS_BSW_00457]	Callback functions of Application software components shall be invoked by the Basis SW	[SWS_NvM_00467] [SWS_NvM_00468] [SWS_NvM_00469] [SWS_NvM_00539] [SWS_NvM_00540]
[SRS_LIBS_08533]	No description	[SWS_NvM_00454] [SWS_NvM_00460] [SWS_NvM_00540] [SWS_NvM_00764]
[SRS_LIBS_08535]	No description	[SWS_NvM_00018] [SWS_NvM_00253] [SWS_NvM_00461]
[SRS_MEM_00020]	The NVRAM manager shall provide functionality to read out the status of read/write operations	[SWS_NvM_00888] [SWS_NvM_00889] [SWS_NvM_00890] [SWS_NvM_00891] [SWS_NvM_00892] [SWS_NvM_00949]
[SRS_MEM_00125]	For each block a notification shall be configurable	[SWS_NvM_00890] [SWS_NvM_00891] [SWS_NvM_00892] [SWS_NvM_00949]
[SRS_MEM_00136]	The NVRAM manager shall provide functionality for determining updates of data associated with an NVRAM Block during runtime	[SWS_NvM_00888] [SWS_NvM_00889]
[SRS_MEM_08533]	The NVRAM manager shall provide a service to check and load those NVRAM blocks, configured to have a permanent RAM data block to RAM	[SWS_NvM_00888] [SWS_NvM_00889]
[SRS_MEM_08535]	The NVRAM manager shall provide a function, which triggers update of integrity information and saving of RAM data blocks to NV memory	[SWS_NvM_00888] [SWS_NvM_00889]
[SRS_Mem_00011]	The NVRAM manager shall be independent from its underlying memory hardware.	[SWS_NvM_00157]
[SRS_Mem_00013]	The NVRAM manager shall provide a mechanism to handle multiple, concurrent read / write requests	[SWS_NvM_00162] [SWS_NvM_00698] [SWS_NvM_00699]
[SRS_Mem_00016]	The NVRAM manager shall provide functionality to read out data associated with an NVRAM block from the non-volatile memory	[SWS_NvM_00010] [SWS_NvM_00051] [SWS_NvM_00122] [SWS_NvM_00195] [SWS_NvM_00196] [SWS_NvM_00454] [SWS_NvM_00629] [SWS_NvM_00764] [SWS_NvM_00765] [SWS_NvM_00766] [SWS_NvM_00825] [SWS_NvM_00898] [SWS_NvM_00899]

Requirement	Description	Satisfied by
[SRS_Mem_00017]	The NVRAM manager shall provide functionality to store data associated with an NVRAM block in the non-volatile memory	[SWS_NvM_00051] [SWS_NvM_00122] [SWS_NvM_00210] [SWS_NvM_00410] [SWS_NvM_00455] [SWS_NvM_00622] [SWS_NvM_00793] [SWS_NvM_00794] [SWS_NvM_00897] [SWS_NvM_00900] [SWS_NvM_00901]
[SRS_Mem_00018]	The NVRAM manager shall provide functionality to restore an NVRAM block's associated data from ROM defaults	[SWS_NvM_00012] [SWS_NvM_00051] [SWS_NvM_00122] [SWS_NvM_00266] [SWS_NvM_00267] [SWS_NvM_00435] [SWS_NvM_00456] [SWS_NvM_00813] [SWS_NvM_00814] [SWS_NvM_00816] [SWS_NvM_00817] [SWS_NvM_00893] [SWS_NvM_00894] [SWS_NvM_00902] [SWS_NvM_00903] [SWS_NvM_00951]
[SRS_Mem_00020]	The NVRAM manager shall provide functionality to read out the status of read/write operations	[SWS_NvM_00015] [SWS_NvM_00451] [SWS_NvM_00895] [SWS_NvM_00896]
[SRS_Mem_00027]	The NVRAM manager shall provide an implicit way of accessing blocks in the NVRAM and in the shared memory (RAM).	[SWS_NvM_00442]
[SRS_Mem_00030]	The NVRAM manager shall implement mechanisms for consistency/integrity checks of data saved in NVRAM	[SWS_NvM_00164] [SWS_NvM_00897]
[SRS_Mem_00034]	Write accesses of the NVRAM manager to persistent memory shall be executed quasi-parallel to normal operation of the ECU	[SWS_NvM_00162]
[SRS_Mem_00038]	Treatable errors shall not affect other software components	[SWS_NvM_00748] [SWS_NvM_00825] [SWS_NvM_00910] [SWS_NvM_00911] [SWS_NvM_00948]
[SRS_Mem_00041]	Each application shall be enabled to declare the memory requirements at configuration time	[SWS_NvM_00051] [SWS_NvM_00122]
[SRS_Mem_00125]	For each block a notification shall be configurable	[SWS_NvM_00463]
[SRS_Mem_00127]	The NVRAM manager shall allow enabling/disabling a write protection for each NVRAM block individually	[SWS_NvM_00016] [SWS_NvM_00450] [SWS_NvM_00748]
[SRS_Mem_00129]	The NVRAM manager shall repair data in blocks of management type 'NVRAM redundant'	[SWS_NvM_00165] [SWS_NvM_00582]
[SRS_Mem_00135]	The NVRAM manager shall have an unique configuration identifier	[SWS_NvM_00034]

Requirement	Description	Satisfied by
[SRS_Mem_00136]	The NVRAM manager shall provide functionality for determining updates of data associated with an NVRAM Block during runtime	[SWS_NvM_00849] [SWS_NvM_00850] [SWS_NvM_00852] [SWS_NvM_00853] [SWS_NvM_00854] [SWS_NvM_00906] [SWS_NvM_00909]
[SRS_Mem_00137]	The NVRAM manager shall provide a service for auto-validating NVRAM blocks	[SWS_NvM_00855] [SWS_NvM_00856] [SWS_NvM_00857] [SWS_NvM_00858] [SWS_NvM_00859] [SWS_NvM_00860] [SWS_NvM_00861] [SWS_NvM_00862] [SWS_NvM_00863]
[SRS_Mem_08000]	The NVRAM manager shall be able to access multiple non-volatile memory devices	[SWS_NvM_00051] [SWS_NvM_00123] [SWS_NvM_00442]
[SRS_Mem_08007]	The NVRAM manager shall provide a service for the selection of valid dataset NV blocks	[SWS_NvM_00448]
[SRS_Mem_08009]	The NVRAM Manager shall allow a static configuration of a default write protection (on/off) for each NVRAM block	[SWS_NvM_00325] [SWS_NvM_00326] [SWS_NvM_00577]
[SRS_Mem_08010]	The NVRAM manager shall copy the ROM default data to the data area of the corresponding RAM block if it can not read data from NV into RAM	[SWS_NvM_00171] [SWS_NvM_00172]
[SRS_Mem_08011]	The NVRAM manager shall provide a service to invalidate a block of data in the non-volatile memory	[SWS_NvM_00421] [SWS_NvM_00459]
[SRS_Mem_08014]	The NVRAM manager shall allow a non-continuous RAM block allocation in the global RAM area	[SWS_NvM_00051] [SWS_NvM_00122] [SWS_NvM_00442]
[SRS_Mem_08015]	Some of the NV Blocks in the NVRAM shall never be erased nor be replaced with the default ROM data after first initialization	[SWS_NvM_00397]
[SRS_Mem_08534]	The NVRAM manager shall support two classes of RAM data blocks	[SWS_NvM_00904]
[SRS_Mem_08540]	The NVRAM manager shall provide a function for aborting the shutdown process	[SWS_NvM_00019] [SWS_NvM_00458]
[SRS_Mem_08541]	The NVRAM manager shall guarantee that an accepted write request will be processed	[SWS_NvM_00208] [SWS_NvM_00384] [SWS_NvM_00472] [SWS_NvM_00622] [SWS_NvM_00748] [SWS_NvM_00798]
[SRS_Mem_08542]	The NVRAM manager shall provide a prioritization for job processing order	[SWS_NvM_00032] [SWS_NvM_00378] [SWS_NvM_00564]
[SRS_Mem_08544]	The NVRAM manager shall provide a service to erase the NV block(s) associated with an NVRAM block	[SWS_NvM_00415] [SWS_NvM_00457]

Requirement	Description	Satisfied by
[SRS_Mem_08545]	The NVRAM Manager shall provide a service for marking the permanent RAM data block of an NVRAM block valid	[SWS_NvM_00241] [SWS_NvM_00405] [SWS_NvM_00453] [SWS_NvM_00906] [SWS_NvM_00909]
[SRS_Mem_08546]	It shall be possible to protect permanent RAM data blocks against data loss due to reset	[SWS_NvM_00240] [SWS_NvM_00548]
[SRS_Mem_08547]	The NVRAM Manager shall be able to distinguish between explicitly invalidated and inconsistent data	[SWS_NvM_00132] [SWS_NvM_00164] [SWS_NvM_00165] [SWS_NvM_00174] [SWS_NvM_00571]
[SRS_Mem_08548]	The NVRAM Manager shall request default data from the application	[SWS_NvM_00629] [SWS_NvM_00700] [SWS_NvM_00893] [SWS_NvM_00894]
[SRS_Mem_08549]	The NVRAM manager shall provide functionality to automatically initialize RAM data blocks after a software update	[SWS_NvM_00171]
[SRS_Mem_08550]	The NVRAM Manager shall provide a service for marking permanent RAM data blocks as modified/unmodified	[SWS_NvM_00344] [SWS_NvM_00345] [SWS_NvM_00696] [SWS_NvM_00906] [SWS_NvM_00909]
[SRS_Mem_08554]	The NVRAM manager shall retry read and write operations on NVRAM blocks if they have not succeeded up to a configurable number of times	[SWS_NvM_00213] [SWS_NvM_00526] [SWS_NvM_00527] [SWS_NvM_00529] [SWS_NvM_00581] [SWS_NvM_00804] [SWS_NvM_00897] [SWS_NvM_00907] [SWS_NvM_00908]
[SRS_Mem_08555]	The NVRAM manager shall provide mechanisms for static verification of the block identifier when reading an NVRAM block	[SWS_NvM_00523] [SWS_NvM_00524] [SWS_NvM_00593]
[SRS_Mem_08556]	The NVRAM manager shall provide a mechanism for verification of the written block data by again reading and comparing it	[SWS_NvM_00527] [SWS_NvM_00528] [SWS_NvM_00529] [SWS_NvM_00897]
[SRS_Mem_08558]	The NVRAM manager shall provide a mechanism to remove all unprocessed requests associated with a NVRAM block	[SWS_NvM_00458]
[SRS_Mem_08560]	Each NVRAM block shall be configurable for shared access	[SWS_NvM_00535] [SWS_NvM_00536]
[SWS_BSW_-00047]	No description	[SWS_NvM_00447]
[SWS_NvM_08541]	No description	[SWS_NvM_00897]

7 Functional specification

7.1 Basic architecture guidelines

7.1.1 Layer structure

The figure below shows the communication interaction of module NvM.

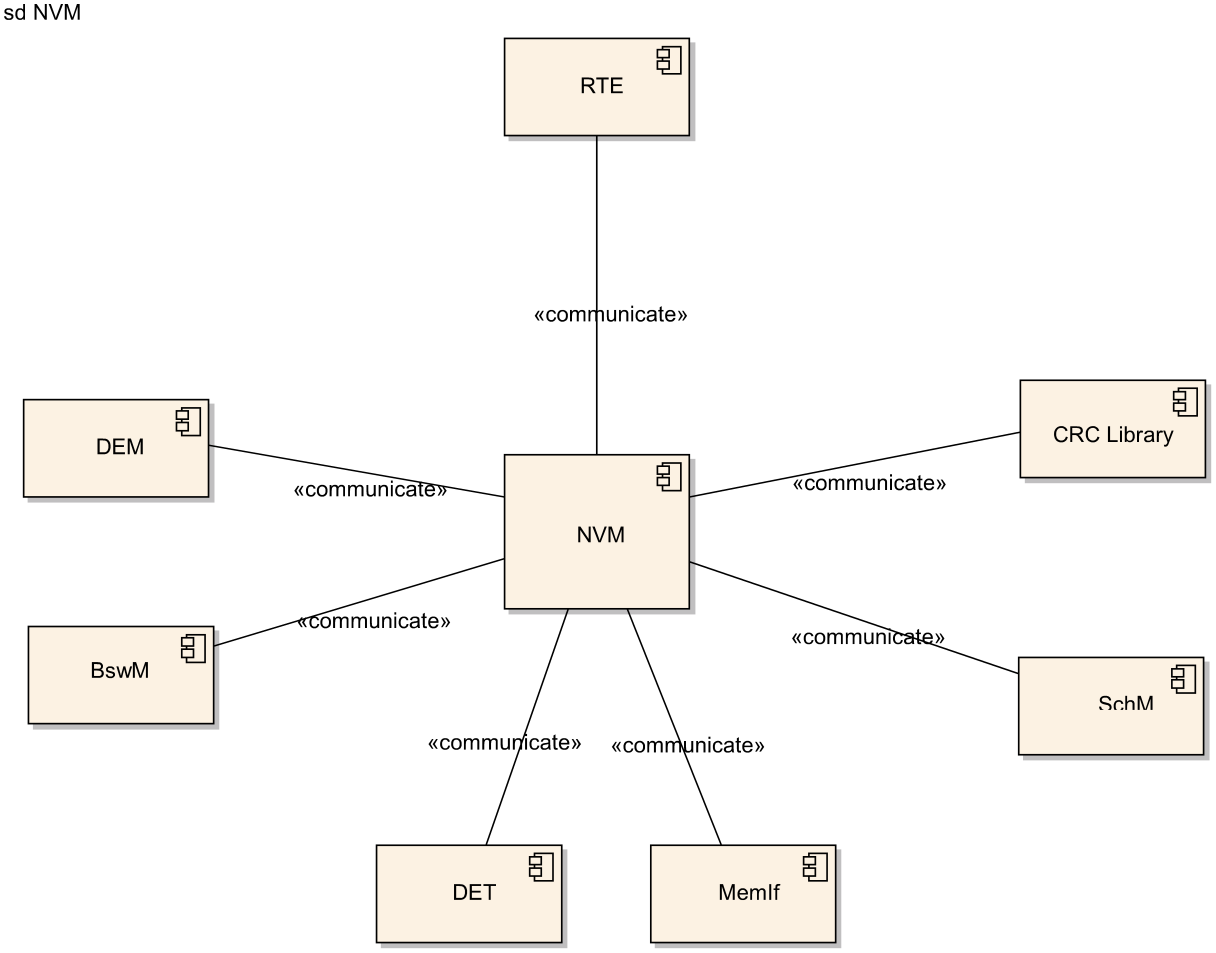

Figure 7.1: NVRAM Manager interactions overview

7.1.2 Addressing scheme for the memory hardware abstraction

[SWS_NvM_00051] ⌈The Memory Abstraction Interface, the underlying Flash EEPROM Emulation and EEPROM Abstraction Layer provide the NvM module with a virtual linear 32bit address space which is composed of a 16bit block number and a 16bit block address offset.⌋*(SRS_Mem_00041, SRS_Mem_08000, SRS_Mem_08014, SRS_Mem_00016, SRS_Mem_00017, SRS_Mem_00018)*

Hint: According to [SWS_NvM_00051], the NvM module allows for a (theoretical) maximum of 65536 logical blocks, each logical block having a (theoretical) maximum size of 64 Kbytes.

[SWS_NvM_00122] ⌈The NvM module shall further subdivide the 16bit Fee/Ea block number into the following parts:

- NV block base number (NVM_NV_BLOCK_BASE_NUMBER) with a bit width of (16 -NVM_DATASET_SELECTION_BITS)
- Data index with a bit width of (NVM_DATASET_SELECTION_BITS)

⌋*(SRS_Mem_00041, SRS_Mem_08014, SRS_Mem_00016, SRS_Mem_00017, SRS_Mem_00018)*

[SWS_NvM_00343] ⌈Handling/addressing of redundant NVRAM blocks shall be done towards the memory hardware abstraction in the same way like for dataset NVRAM blocks, i.e. the redundant NV blocks shall be managed by usage of the configuration parameter NvMDatasetSelectionBits.⌋*()*

[SWS_NvM_00123] ⌈The NV block base number (NVM_NV_BLOCK_BASE_NUMBER) shall be located in the most significant bits of the Fee/Ea block number.⌋*(SRS_Mem_08000)*

[SWS_NvM_00442] ⌈The configuration tool shall configure the block identifiers.⌋ *(SRS_Mem_08000, SRS_Mem_00027, SRS_Mem_08014)*

[SWS_NvM_00443] ⌈The NvM module shall not modify the configured block identifiers.⌋*()*

7.1.2.1 Examples

To clarify the previously described addressing scheme which is used for NVRAM manager ↔ memory hardware abstraction interaction, the following examples shall help to understand the correlations between the configuration parameters NvMNvBlockBase Number, NvMDatasetSelectionBits on NVRAM manager side and EA_BLOCK_NUMBER / FEE_BLOCK_NUMBER on memory hardware abstraction side (see 10.2.3).

For the given examples A and B a simple formula is used:

FEE/EA_BLOCK_NUMBER = (NvMNvBlockBaseNumber << NvMDatasetSelection Bits) + DataIndex.

Example A:

- The configuration parameter NvMDatasetSelectionBits is configured to be 2. This leads to the result that 14 bits are available as range for the configuration parameter NvMNvBlockBaseNumber.
- Range of NvMNvBlockBaseNumber: 0x1..0x3FFE

- Range of data index: 0x0..0x3(=2^NvMDatasetSelectionBits-1)
- Range of FEE_BLOCK_NUMBER/EA_BLOCK_NUMBER: 0x4..0xFFFB

With this configuration the FEE/EA_BLOCK_NUMBER computes using the formula mentioned before should look like in the examples below:

For a native NVRAM block with NvMNvBlockBaseNumber = 2:

- NV block is accessed with FEE/EA_BLOCK_NUMBER = 8

For a redundant NVRAM block with NvMNvBlockBaseNumber = 3:

- 1st NV block with data index 0 is accessed with FEE/EA_BLOCK_NUMBER = 12
- 2nd NV block with data index 1 is accessed with FEE/EA_BLOCK_NUMBER = 13

For a dataset NVRAM block with NvMNvBlockBaseNumber = 4, NvMNvBlockNum = 3:

- NV block #0 with data index 0 is accessed with FEE/EA_BLOCK_NUMBER = 16
- NV block #1 with data index 1 is accessed with FEE/EA_BLOCK_NUMBER = 17
- NV block #2 with data index 2 is accessed with FEE/EA_BLOCK_NUMBER = 18

Example B:

- The configuration parameter NvMDatasetSelectionBits is configured to be 4. This leads to the result that 12 bits are available as range for the configuration parameter NvMNvBlockBaseNumber.
- Range of NvMNvBlockBaseNumber: 0x1..0xFFE
- Range of data index: 0x0..0xF(=2^NvMDatasetSelectionBits-1)
- Range of FEE/EA Block Number: 0x10..0xFFEF

7.1.3 Basic storage objects

7.1.3.1 NV block

[SWS_NvM_00125] ⌈The NV block is a basic storage object and represents a memory area consisting of NV user data and (optionally) a CRC value and (optionally) a NV block header.⌋ *()*

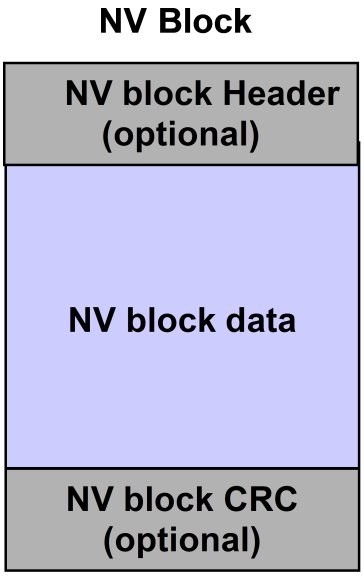

Figure 7.2: NV Block layout

Note: This figure does not show the physical memory layout of an NV block. Only the logical clustering is shown.

7.1.3.2 RAM block

[SWS_NvM_00126] ⌈The RAM block is a basic storage object and represents an area in RAM consisting of user data and (optionally) a CRC value and (optionally) a NV block header.⌋*()*

[SWS_NvM_00127] ⌈Restrictions on CRC usage on RAM blocks. CRC is only available if the corresponding NV block(s) also have a CRC. CRC has to be of the same type as that of the corresponding NV block(s) (see 10.2.3).⌋*()*

[SWS_NvM_00129] ⌈The user data area of a RAM block can reside in a different RAM address location (global data section) than the state of the RAM block.⌋*()*

[SWS_NvM_00130] ⌈The data area of a RAM block shall be accessible from NVRAM Manager and from the application side (data passing from/to the corresponding NV block).⌋*()*

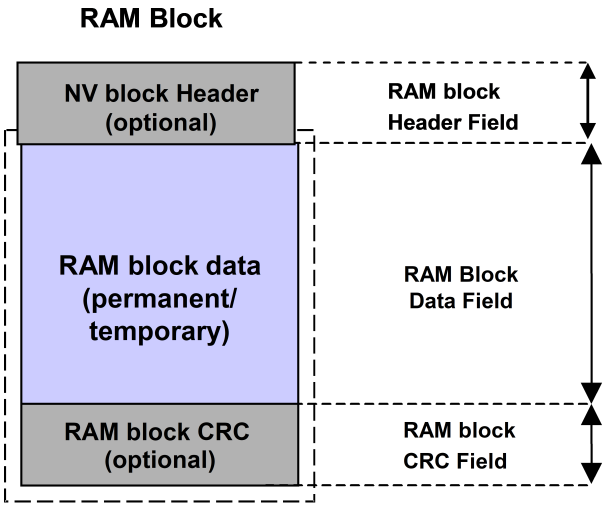

Figure 7.3: RAM Block layout

Note: This figure does not show the physical memory layout of a RAM block. Only the logical clustering is shown.

As the NvM module doesn't support alignment, this could be managed by configuration, i.e. the block length could be enlarged by adding padding to meet alignment requirements.

[SWS_NvM_00373] ⌈The RAM block data shall contain the permanently or temporarily assigned user data.⌋*()*

[SWS_NvM_00370] ⌈In case of permanently assigned user data, the address of the RAM block data is known during configuration time.⌋*()*

[SWS_NvM_00372] ⌈In case of temporarily assigned user data, the address of the RAM block data is not known during configuration time and will be passed to the NvM module during runtime.⌋*()*

[SWS_NvM_00088] ⌈It shall be possible to allocate each RAM block without address constraints in the global RAM area. The whole number of configured RAM blocks needs not be located in a continuous address space.⌋*()*

7.1.3.3 ROM block

[SWS_NvM_00020] ⌈The ROM block is a basic storage object, resides in the ROM (FLASH) and is used to provide default data in case of an empty or damaged NV block.

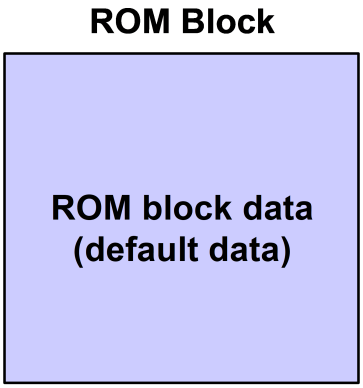

Figure 7.4: ROM block layout

⌋()

7.1.3.4 Administrative block

[SWS_NvM_00134] ⌈The Administrative block shall be located in RAM and shall contain a block index which is used in association with Dataset NV blocks. Additionally, attribute/error/status information of the corresponding NVRAM block shall be contained.⌋ ()

[SWS_NvM_00128] ⌈The NvM module shall use state information of the permanent RAM block or of the RAM mirror in the NvM module in case of explicit syncronization (invalid/valid) to determine the validity of the permanent RAM block user data.⌋()

[SWS_NvM_00132] ⌈The RAM block state "invalid" indicates that the data area of the respective RAM block is invalid. The RAM block state "valid" indicates that the data area of the respective RAM block is valid.⌋(*SRS_Mem_08547*)

[SWS_NvM_00133] ⌈The value of "invalid" shall be represented by all other values except "valid".⌋()

[SWS_NvM_00135] ⌈The Administrative block shall be invisible for the application and is used exclusively by the NvM module for security and administrative purposes of the RAM block and the NVRAM block itself.⌋()

[SWS_NvM_00054] ⌈The NvM module shall use an attribute field to manage the NV block write protection in order to protect/unprotect a NV block data field.⌋()

[SWS_NvM_00136] ⌈The NvM module shall use an error/status field to manage the error/status value of the last request.⌋()

7.1.3.5 NV Block Header

[SWS_NvM_00522] ⌈The NV Block header shall be included first in the NV Block, if the mechanism Static Block ID is enabled.

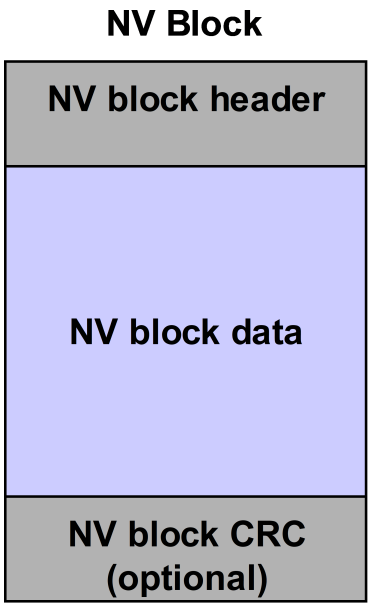

Figure 7.5: NV block layout with Static Block ID enabled

⌋()

7.1.4 Block management types

7.1.4.1 Block management types overview

[SWS_NvM_00137] ⌈The following types of NVRAM storage shall be supported by the NvM module implementation:

- NVM_BLOCK_NATIVE
- NVM_BLOCK_REDUNDANT
- NVM_BLOCK_DATASET

⌋()

[SWS_NvM_00557] ⌈NVM_BLOCK_NATIVE type of NVRAM storage shall consist of the following basic storage objects:

- NV Blocks: 1
- RAM Blocks: 1
- ROM Blocks: 0..1
- Administrative Blocks:1

⌋()

[SWS_NvM_00558] ⌈NVM_BLOCK_REDUNDANT type of NVRAM storage shall consist of the following basic storage objects:

- NV Blocks: 2
- RAM Blocks: 1
- ROM Blocks: 0..1
- Administrative Blocks:1

⌋()

[SWS_NvM_00559] ⌈NVM_BLOCK_DATASET type of NVRAM storage shall consist of the following basic storage objects:

- NV Blocks: 1..(m<256)*
- RAM Blocks: 1
- ROM Blocks: 0..n
- Administrative Blocks:1
- * The number of possible datasets depends on the configuration parameter Nv MDatasetSelectionBits.

⌋()

7.1.4.2 NVRAM block structure

[SWS_NvM_00138] ⌈The NVRAM block shall consist of the mandatory basic storage objects NV block, RAM block and Administrative block.⌋()

[SWS_NvM_00139] ⌈The basic storage object ROM block is optional.⌋()

[SWS_NvM_00140] ⌈The composition of any NVRAM block is fixed during configuration by the corresponding NVRAM block descriptor.⌋()

[SWS_NvM_00141] ⌈All address offsets are given relatively to the start addresses of RAM or ROM in the NVRAM block descriptor. The start address is assumed to be zero.⌋()

Hint: A device specific base address or offset will be added by the respective device driver if needed.

For details of the NVRAM block descriptor see chapter 7.1.4.3.

7.1.4.3 NVRAM block descriptor table

[SWS_NvM_00069] ⌈A single NVRAM block to deal with will be selected via the NvM module API by providing a subsequently assigned Block ID.⌋()

[SWS_NvM_00143] ⌈All structures related to the NVRAM block descriptor table and their addresses in ROM (FLASH) have to be generated during configuration of the Nv M module.⌋*()*

7.1.4.4 Native NVRAM block

The Native NVRAM block is the simplest block management type. It allows storage to/ retrieval from NV memory with a minimum of overhead.

[SWS_NvM_00000] ⌈The Native NVRAM block consists of a single NV block, RAM block and Administrative block.⌋*()*

7.1.4.5 Redundant NVRAM block

In addition to the Native NVRAM block, the Redundant NVRAM block provides enhanced fault tolerance, reliability and availability. It increases resistance against data corruption.

[SWS_NvM_00001] ⌈The Redundant NVRAM block consists of two NV blocks, a RAM block and an Administrative block.⌋*()*

The following figure reflects the internal structure of a redundant NV block:

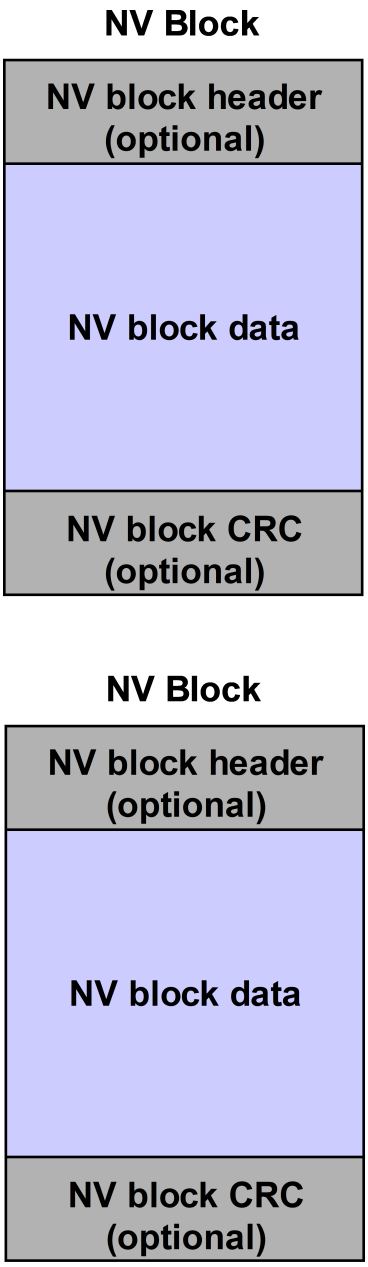

Figure 7.6: Redundant NVRAM block layout

Note: This figure does not show the physical NV memory layout of a redundant NVRAM block. Only the logical clustering is shown.

[SWS_NvM_00531] ⌈In case one NV Block associated with a Redundant NVRAM block is deemed invalid (e.g. during read), an attempt shall be made to recover the NV Block using data from the incorrupt NV Block.⌋*()*

[SWS_NvM_00546] ⌈In case the recovery fails then this shall be reported to the DEM using the code NVM_E_LOSS_OF_REDUNDANCY.⌋*()*

Note: "Recovery" denotes the re-establishment of redundancy. This usually means writing the recovered data back to the NV Block.

7.1.4.6 Dataset NVRAM block

The Dataset NVRAM block is an array of equally sized data blocks (NV/ROM). The application can at one time access exactly one of these elements.

[SWS_NvM_00006] ⌈The Dataset NVRAM block consists of multiple NV user data, (optionally) CRC areas, (optional) NV block headers, a RAM block and an Administrative block.⌋*()*

[SWS_NvM_00144] ⌈The index position of the dataset is noticed via a separated field in the corresponding Administrative block.⌋*()*

[SWS_NvM_00374] ⌈The NvM module shall be able to read all assigned NV blocks.⌋*()*

[SWS_NvM_00375] ⌈The NvM module shall only be able to write to all assigned NV blocks if (and only if) write protection is disabled.⌋*()*

[SWS_NvM_00146] ⌈If the basic storage object ROM block is selected as optional part, the index range which normally selects a dataset is extended to the ROM to make it possible to select a ROM block instead of a NV block. The index covers all NV/ROM blocks which may build up the NVRAM Dataset block.⌋*()*

[SWS_NvM_00376] ⌈The NvM module shall be able to only read optional ROM blocks (default datasets).⌋*()*

[SWS_NvM_00377] ⌈The NvM module shall treat a write to a ROM block like a write to a protected NV block.⌋*()*

[SWS_NvM_00444] ⌈The total number of configured datasets (NV+ROM blocks) shall be in the range of 1..255.⌋*()*

[SWS_NvM_00445] ⌈In case of optional ROM blocks, data areas with an index from 0 up to NvMNvBlockNum - 1 represent the NV blocks with their CRC in the NV memory. Data areas with an index from NvMNvBlockNum up to NvMNvBlockNum + NvMRom BlockNum - 1 represent the ROM blocks.⌋*()*

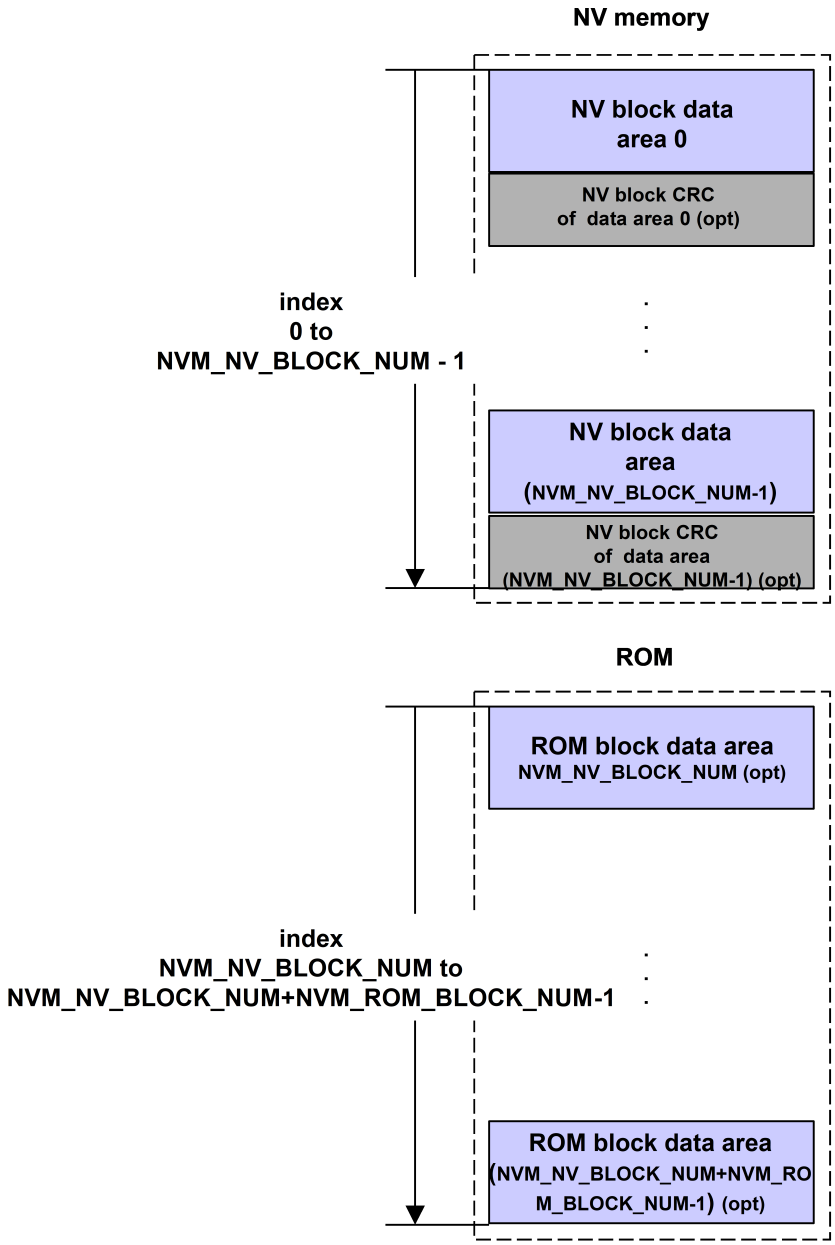

Figure 7.7: Dataset NVRAM block layout

Note: This figure does not show the physical NV memory layout of a Dataset NVRAM block. Only the logical clustering is shown.

7.1.4.7 NVRAM Manager API configuration classes

[SWS_NvM_00149] ⌈To have the possibility to adapt the NvM module to limited hardware resources, three different API configuration classes shall be defined:

- API configuration class 3: All specified API calls are available. A maximum of functionality is supported.

- API configuration class 2: An intermediate set of API calls is available.
- API configuration class 1: Especially for matching systems with very limited hardware resources this API configuration class offers only a minimum set of API calls which are required in any case.

⌋()

[SWS_NvM_00560] ⌈API configuration class 3 shall consist of the following API:

- Type 1:
- NvM_SetDataIndex(...)
- NvM_GetDataIndex(...)
- NvM_SetBlockProtection(...)
- NvM_GetErrorStatus(...)
- NvM_SetRamBlockStatus(...)
- NvM_SetBlockLockStatus()
- Type 2:
- NvM_ReadBlock(...)
- NvM_WriteBlock(...)
- NvM_RestoreBlockDefaults(...)
- NvM_EraseNvBlock(...)
- NvM_InvalidateNvBlock(...)
- NvM_CancelJobs(...)
- NvM_ReadPRAMBlock(...)
- NvM_WritePRAMBlock(...)
- NvM_RestorePRAMBlockDefaults(...)
- Type 3:
- NvM_ReadAll(...)
- NvM_WriteAll(...)
- NvM_CancelWriteAll(...)
- NvM_ValidateAll(...)
- NvM_FirstInitAll(...)
- Type 4:
- NvM_Init(...)

⌋()

[SWS_NvM_00561] ⌈API configuration class 2 shall consist of the following API:

- Type 1:
- NvM_SetDataIndex(...)
- NvM_GetDataIndex(...)
- NvM_GetErrorStatus(...)
- NvM_SetRamBlockStatus(...)
- NvM_SetBlockLockStatus(...)
- Type 2:
- NvM_ReadBlock(...)
- NvM_WriteBlock(...)
- NvM_RestoreBlockDefaults(...)
- NvM_CancelJobs(...)
- NvM_ReadPRAMBlock(...)
- NvM_WritePRAMBlock(...)
- NvM_RestorePRAMBlockDefaults(...)
- Type 3:
- NvM_ReadAll(...)
- NvM_WriteAll(...)
- NvM_CancelWriteAll(...)
- NvM_ValidatedAll(...)
- Type 4:
- NvM_Init(...)

⌋()

[SWS_NvM_00562] ⌈API configuration class 1 shall consist of the following API:

- Type 1:
- NvM_GetErrorStatus(...)
- NvM_SetRamBlockStatus(...)
- NvM_SetBlockLockStatus(...)
- Type 2:

- –
- Type 3:
- NvM_ReadAll(...)
- NvM_WriteAll(...)
- NvM_CancelWriteAll(...)
- Type 4:
- NvM_Init(...)
- Note: For API configuration class 1 no queues are needed, no immediate data can be written. Furthermore the API call NvM_SetRamBlockStatus is only available if configured by NvMSetRamBlockStatusApi.

⌋()

[SWS_NvM_00365] ⌈Within API configuration class 1, the block management type NVM_BLOCK_DATASET is not supported.⌋()

For information regarding the definition of Type 1. . . 4 refer to chapter 8.6.

[SWS_NvM_00150] ⌈The NvM module shall only contain that code that is needed to handle the configured block types.⌋()

7.1.5 Scan order / priority scheme

[SWS_NvM_00032] ⌈The NvM module shall support a priority based job processing.⌋ (*SRS_Mem_08542*)

[SWS_NvM_00564] ⌈By configuration parameter NvMJobPrioritization priority based job processing shall be enabled/disabled (see 10.2.2).⌋(*SRS_Mem_08542*)

[SWS_NvM_00378] ⌈In case of priority based job processing order, the NvM module shall use two queues, one for immediate write jobs (crash data) another for all other jobs (including immediate read/erase jobs).⌋(*SRS_Mem_08542*)

[SWS_NvM_00379] ⌈If priority based job processing is disabled via configuration, the NvM module shall not support immediate write jobs. In this case, the NvM module processes all jobs in FCFS order.⌋()

[SWS_NvM_00380] ⌈The job queue length for multi block requests originating from any of the NvM_ReadAll, NvM_ValidateAll, NvM_FirstInitAll and NvM_WriteAll APIs shall be one (only one multi block job is queued).⌋()

[SWS_NvM_00381] ⌈The NvM module shall not interrupt jobs originating from the Nv M_ReadAll request by other requests.⌋()

Note: The only exception to the rule given in [SWS_NvM_00381], [SWS_NvM_00567] is a write job with immediate priority which shall preempt the running read / write job

[SWS_NvM_00182]. The preempted job shall subsequently be resumed / restarted by the NvM module.

[SWS_NvM_00567] ⌈The NvM module shall not interrupt jobs originating from the NvM_WriteAll request by other requests.⌋*()*

[SWS_NvM_00568] ⌈The NvM module shall rather queue read jobs that are requested during an ongoing NvM_ReadAll request and executed them subsequently.⌋*()*

[SWS_NvM_00569] ⌈The NvM module shall rather queue write jobs that are requested during an ongoing NvM_WriteAll request and executed them subsequently.⌋*()*

[SWS_NvM_00725] ⌈The NvM module shall rather queue write jobs that are requested during an ongoing NvM_ReadAll request and executed them subsequently.⌋*()*

[SWS_NvM_00726] ⌈The NvM module shall rather queue read jobs that are requested during an ongoing NvM_WriteAll request and executed them subsequently.⌋*()*

Note: The NvM_WriteAll request can be aborted by calling NvM_CancelWriteAll. In this case, the current block is processed completely but no further blocks are written [SWS_NvM_00238].

Hint: It shall be allowed to dequeue requests, if they became obsolete by completion of the regarding NVRAM block.

[SWS_NvM_00570] ⌈The preempted job shall subsequently be resumed / restarted by the NvM module. This behavior shall apply for single block requests as well as for multi block requests.⌋*()*

7.2 General behavior

7.2.1 Functional requirements

[SWS_NvM_00383] ⌈For each asynchronous request, a notification of the caller after completion of the job shall be a configurable option.⌋*()*

[SWS_NvM_00384] ⌈The NvM module shall provide a callback interface [SWS_NvM_00113].⌋*(SRS_Mem_08541)*

Hint: The NvM module's environment shall access the non-volatile memory via the NvM module only. It shall not be allowed for any module (except for the NvM module) to access the non-volatile memory directly.

[SWS_NvM_00038] ⌈The NvM module only provides an implicit way of accessing blocks in the NVRAM and in the shared memory (RAM). This means, the NvM module copies one or more blocks from NVRAM to the RAM and the other way round.⌋*()*

[SWS_NvM_00692] ⌈The application accesses the RAM data directly, with respect to given restrictions (e.g. synchronization).⌋*()*

[SWS_NvM_00385] ⌈The NvM module shall queue all asynchronous "single block" read/write/control requests if the block with its specific ID is not already queued or currently in progress (multitasking restrictions).⌋*()*

[SWS_NvM_00386] ⌈The NvM module shall accept multiple asynchronous "single block" requests as long as no queue overflow occurs.⌋*()*

[SWS_NvM_00155] ⌈The highest priority request shall be fetched from the queues by the NvM module and processed in a serialized order.⌋*()*

[SWS_NvM_00040] ⌈The NvM module shall implement implicit mechanisms for consistency / integrity checks of data saved in NV memory [SWS_NvM_00165].⌋*()*

Depending on implementation of the memory stack, callback routines provided and/or invoked by the NvM module may be called in interrupt context.

Hint: The NvM module providing routines called in interrupt context has therefore to make sure that their runtime is reasonably short.

[SWS_NvM_00085] ⌈If there is no default ROM data available at configuration time or no callback defined by NvMInitBlockCallback then the application shall be responsible for providing the default initialization data. In this case, the application has to use NvM_GetErrorStatus() to be able to distinguish between first initialization and corrupted data (see 10.2.3).⌋*()*

[SWS_NvM_00387] ⌈During processing of NvM_ReadAll, the NvM module shall be able to detect corrupted RAM data by performing a checksum calculation. [ECUC_NvM_00476].⌋*()*

[SWS_NvM_00226] ⌈During processing of NvM_ReadAll, the NvM module shall be able to detect invalid RAM data by testing the validity of a data within the administrative block [ECUC_NvM_00476].⌋*()*

[SWS_NvM_00388] ⌈During startup phase and normal operation of NvM_ReadAll and if the NvM module has detected an unrecoverable error within the NV block, the NvM module shall copy default data (if configured) to the corresponding RAM block.⌋*()*

[SWS_NvM_00332] ⌈To make use of the OS services, the NvM module shall only use the BSW scheduler instead of directly making use of OS objects and/or related OS services.⌋*(SRS_BSW_00429)*

[SWS_NvM_00985] ⌈The NvM module shall use the internal mirror as a buffer for all operations that read and write the RAM block of the NVRAM blocks with configured permanent RAM (or RAM passed by API parameter) for which the RAM (start) is not aligned to the NvMBufferAlignmentValue.⌋*()*

7.2.2 Design notes

7.2.2.1 NVRAM manager startup

[SWS_NvM_00693] ⌈NvM_Init shall be invoked by the BSW Mode Manager exclusively.⌋*()*

[SWS_NvM_00091] ⌈Due to strong constraints concerning the ECU startup time, the NvM_Init request shall not contain the initialization of the configured NVRAM blocks.⌋*()*

[SWS_NvM_00157] ⌈The NvM_Init request shall not be responsible to trigger the initialization of underlying drivers and memory hardware abstraction. This shall also be handled by the BSW Mode Manager.⌋*(SRS_Mem_00011)*

[SWS_NvM_00158] ⌈The initialization of the RAM data blocks shall be done by another request, namely NvM_ReadAll.⌋*()*

NvM_ReadAll shall be called exclusively by BSW Mode Manager.

[SWS_NvM_00694] ⌈Software components which use the NvM module shall be responsible for checking global error/status information resulting from the NvM module startup. The BSW Mode Manager shall use polling by using NvM_GetErrorStatus [SWS_NvM_00015] (reserved block ID 0) or callback notification (configurable option NvM_MultiBlockCallback 10.2.2) to derive global error/status information resulting from startup. If polling is used, the end of the NVRAM startup procedure shall be detected by the global error/status NVM_REQ_OK or NVM_REQ_NOT_OK (during startup NVM_REQ_PENDING). If callbacks are chosen for notification, software components shall be notified automatically if an assigned NVRAM block has been processed [SWS_NvM_00281].⌋*()*

Note 1: If callbacks are configured for each NVRAM block which is processed within NvM_ReadAll, they can be used by the RTE to start e.g. SW-Cs at an early point of time.

Note 2: To ensure that the DEM is fully operational at an early point of time, i.e. its NV data is restored to RAM, DEM related NVRAM blocks should be configured to have a low ID to be processed first within NvM_ReadAll.

[SWS_NvM_00160] ⌈The NvM module shall not store the currently used Dataset index automatically in a persistent way.

Software components shall check the specific error/status of all blocks they are responsible for by using NvM_GetErrorStatus [SWS_NvM_00015] with specific block IDs to determine the validity of the corresponding RAM blocks.⌋*()*

[SWS_NvM_00695] ⌈For all blocks of the block management type "NVRAM Dataset" [SWS_NvM_00006] the software component shall be responsible to set the proper index position by NvM_SetDataIndex [SWS_NvM_00014]. E.g. the current index position can be stored/maintained by the software component in a unique NVRAM block.

To get the current index position of a "Dataset Block", the software component shall use the NvM_GetDataIndex [SWS_NvM_00021] API call.⌋*()*

7.2.2.2 NVRAM manager shutdown

[SWS_NvM_00092] ⌈The basic shutdown procedure shall be done by the request Nv M_WriteAll [SWS_NvM_00018].⌋*()*

Hint: NvM_WriteAll shall be invoked by the BSW Mode Manager.

7.2.2.3 (Quasi) parallel write access to the NvM module

[SWS_NvM_00162] ⌈The NvM module shall receive the requests via an asynchronous interface using a queuing mechanism. The NvM module shall process all requests serially depending on their priority.⌋*(SRS_Mem_00013, SRS_Mem_00034)*

7.2.2.4 NVRAM block consistency check

[SWS_NvM_00164] ⌈The NvM module shall provide implicit techniques to check the data consistency of NVRAM blocks [ECUC_NvM_00476], [SWS_NvM_00040].⌋ *(SRS_Mem_08547, SRS_Mem_00030)*

[SWS_NvM_00571] ⌈The data consistency check of a NVRAM block shall be done by CRC recalculations of its corresponding NV block(s).⌋*(SRS_Mem_08547)*

[SWS_NvM_00165] ⌈The implicit way of a data consistency check shall be provided by configurable options of the internal functions. The implicit consistency check shall be configurable for each NVRAM block and depends on the configurable parameters Nv MBlockUseCrc and NvMCalcRamBlockCrc (see 10.2.3).⌋*(SRS_Mem_08547, SRS_-Mem_00129)*

[SWS_NvM_00724] ⌈NvMBlockUseCrc should be enabled for NVRAM blocks where NvMWriteBlockOnce = TRUE. NvMBlockWriteProt should be disabled for NVRAM blocks where NvMWriteBlockOnce = TRUE, to enable the user to write data to the NVRAM block in case of CRC check is failed.⌋*()*

[SWS_NvM_00544] ⌈Depending on the configurable parameters NvMBlockUseCrc and NvMCalcRamBlockCrc, NvM module shall allocate memory for the largest CRC used.⌋*()*

Hint: NvM users should not know anything about CRC memory (e.g. size, location) for their data in a RAM block.

7.2.2.5 Error recovery

[SWS_NvM_00047] ⌈The NvM module shall provide techniques for error recovery. The error recovery depends on the NVRAM block management type [SWS_NvM_00001].⌋ *()*

[SWS_NvM_00389] ⌈The NvM module shall provide error recovery on read for every kind of NVRAM block management type by loading of default values.⌋ *()*

[SWS_NvM_00390] ⌈The NvM module shall provide error recovery on read for NVRAM blocks of block management type NVM_BLOCK_REDUNDANT by loading the RAM block with default values.⌋ *()*

[SWS_NvM_00168] ⌈The NvM module shall provide error recovery on write by performing write retries regardless of the NVRAM block management type.⌋ *()*

[SWS_NvM_00169] ⌈The NvM module shall provide read error recovery on startup for all NVRAM blocks with configured RAM block CRC in case of RAM block revalidation failure.⌋ *()*

7.2.2.6 Recovery of a RAM block with ROM data

[SWS_NvM_00171] ⌈The NvM module shall provide implicit and explicit recovery techniques to restore ROM data to its corresponding RAM block in case of unrecoverable data inconsistency of a NV block [SWS_NvM_00387], [SWS_NvM_00226], [SWS_NvM_00388].⌋ *(SRS_Mem_08549, SRS_Mem_08010)*

7.2.2.7 Implicit recovery of a RAM block with ROM default data

[SWS_NvM_00172] ⌈The data content of the corresponding NV block shall remain unmodified during the implicit recovery.⌋ *(SRS_Mem_08010)*

[SWS_NvM_00572] ⌈The implicit recovery shall not be provided during startup (part of NvM_ReadAll), neither by NvM_ReadBlock nor by NvM_ReadPRAMBlock for each NVRAM block when no default data is configured (by the parameter NvMRomBlock DataAddress or NvMInitBlockCallback).⌋ *()*

[SWS_NvM_00573] ⌈The implicit recovery shall not be provided during startup (part of NvM_ReadAll), neither by NvM_ReadBlock nor by NvM_ReadPRAMBlock for each NVRAM block for the following conditions:

- The default data is configured (by the parameter NvMRomBlockDataAddress or the parameter NvMInitBlockCallback).
- The permanent RAM block or the content of the RAM mirror in the NvM module (in case of explicit synchronization) state is valid and CRC (data) is consistent.

⌋ *()*

[SWS_NvM_00574] ⌈The implicit recovery shall not be provided during startup (part of NvM_ReadAll), neither by NvM_ReadBlock nor by NvM_ReadPRAMBlock for each NVRAM block for the following conditions:

- The default data is configured (by the parameter NvMRomBlockDataAddress or the parameter NvMInitBlockCallback).
- The permanent RAM block or the content of the RAM mirror in the NvM module (in case of explicit synchronization) state is invalid and CRC (data) is inconsistent.
- Read attempt from NV succeeds.

⌋*()*

[SWS_NvM_00575] ⌈The implicit recovery shall be provided during startup (part of Nv M_ReadAll) and by NvM_ReadBlock or NvM_ReadPRAMBlock for each NVRAM block for the following conditions:

- The default data is configured (by the parameter NvMRomBlockDataAddress or the parameter NvMInitBlockCallback).
- The permanent RAM block state or the content of the RAM mirror in the NvM module (in case of explicit synchronization) is invalid and CRC (data) is inconsistent.
- Read attempt from NV fails.

⌋*()*

[SWS_NvM_00951] ⌈Implicit recovery shall be provided during NvM_ReadBlock() or NvM_ReadPRAMBlock() requests for NVRAM blocks of type NVM_BLOCK_NATIVE and NVM_BLOCK_REDUNDANT.⌋*(SRS_Mem_00018)*

7.2.2.8 Explicit recovery of a RAM block with ROM default data

[SWS_NvM_00391] ⌈For explicit recovery with ROM block data the NvM module shall provide functions NvM_RestoreBlockDefaults and NvM_RestorePRAMBlockDefaults [SWS_NvM_00012] to restore ROM data to its corresponding RAM block.⌋*()*

[SWS_NvM_00392] ⌈The function NvM_RestoreBlockDefaults and NvM_Restore PRAMBlockDefaults shall remain unmodified the data content of the corresponding NV block.

Hint: The function NvM_RestoreBlockDefaults or NvM_RestorePRAMBlockDefaults shall be used by the application to restore ROM data to the corresponding RAM block every time it is needed.⌋*()*

7.2.2.9 Detection of an incomplete write operation to a NV block

[SWS_NvM_00174] ⌈The detection of an incomplete write operation to a NV block is out of scope of the NvM module. This is handled and detected by the memory hardware abstraction. The NvM module expects to get information from the memory hardware abstraction if a referenced NV block is invalid or inconsistent and cannot be read when requested. SW-Cs may use NvM_InvalidateNvBlock to prevent lower layers from delivering old data.⌋*(SRS_Mem_08547)*

7.2.2.10 Termination of a single block request

[SWS_NvM_00175] ⌈All asynchronous requests provided by the NvM module (except for NvM_CancelWriteAll) shall indicate their result in the designated error/status field of the corresponding Administrative block [SWS_NvM_00000].⌋*()*

[SWS_NvM_00176] ⌈The optional configuration parameter NvMSingleBlockCallback configures the notification via callback on the termination of an asynchronous block request (and for NvM_ReadAll) (see 10.2.3).⌋*()*

Note: In communication with application SW-C, the ECUC configuration parameter Nv MSingleBlockCallback (ECUC_NvM_00506) should be configured to the corresponding Rte_call_<p>_<o> API.

7.2.2.11 Termination of a multi block request

[SWS_NvM_00393] ⌈The NvM module shall use a separate variable to store the result of an asynchronous multi block request (NvM_ReadAll, NvM_WriteAll including NvM_ CancelWriteAll, NvM_ValidateAll).⌋*()*

[SWS_NvM_00394] ⌈The function NvM_GetErrorStatus [SWS_NvM_00015] shall return the most recent error/status information of an asynchronous multi block request (including NvM_CancelWriteAll) in conjunction with a reserved block ID value of 0.⌋*()*

[SWS_NvM_00395] ⌈The result of a multi block request shall represent only a common error/status information.⌋*()*

[SWS_NvM_00396] ⌈The multi block requests provided by the NvM module shall indicate their detailed error/status information in the designated error/status field of each affected Administrative block.⌋*()*

[SWS_NvM_00179] ⌈The optional configuration parameter NvMMultiBlockCallback configures the notification via callback on the termination of an asynchronous multi block request (see 10.2.2).⌋*()*

7.2.2.12 General handling of asynchronous requests/ job processing

[SWS_NvM_00180] ⌈Every time when CRC calculation is processed within a request, the NvM module shall calculate the CRC in multiple steps if the referenced NVRAM block length exceeds the number of bytes configured by the parameter NvMCrcNumOf Bytes.⌋*()*

[SWS_NvM_00351] ⌈For CRC calculation, the NvM module shall use initial values which are published by the CRC module.⌋*()*

[SWS_NvM_00181] ⌈Multiple concurrent single block requests shall be queueable.⌋*()*

[SWS_NvM_00182] ⌈The NvM module shall interrupt asynchronous request/job processing in favor of jobs with immediate priority (crash data).⌋*()*

[SWS_NvM_00184] ⌈If the invocation of an asynchronous function on the NvM module leads to a job queue overflow, the function shall return with E_NOT_OK.⌋*()*

[SWS_NvM_00185] ⌈On successful enqueuing a request, the NvM module shall set the request result of the corresponding NVRAM block to NVM_REQ_PENDING.⌋*()*

[SWS_NvM_00270] ⌈If the NvM module has successfully processed a job, it shall return NVM_REQ_OK as request result.⌋*()*

7.2.2.13 NVRAM block write protection

The NvM module shall offer different kinds of write protection which shall be configurable. Every kind of write protection is only related to the NV part of NVRAM block, i.e. the RAM block data can be modified but not be written to NV memory.

[SWS_NvM_00325] ⌈Enabling/Disabling of the write protection is allowed using NvM_ SetBlockProtection function when the NvMWriteBlockOnce is FALSE regardless of the value (True/False) configured for NvMBlockWriteProt.⌋*(SRS_Mem_08009)*

[SWS_NvM_00577] ⌈Enabling/Disabling of the write protection is not allowed using Nv M_SetBlockProtection function when the NvMWriteBlockOnce is TRUE regardless of the value (True/False) configured for NvMBlockWriteProt.⌋*(SRS_Mem_08009)*

[SWS_NvM_00326] ⌈For all NVRAM blocks configured with NvMBlockWriteProt = TRUE, the NvM module shall enable a default write protection.⌋*(SRS_Mem_08009)*

[SWS_NvM_00578] ⌈The NvM module's environment can explicitly disable the write protection using the NvM_SetBlockProtection function.⌋*()*

[SWS_NvM_00397] ⌈For NVRAM blocks configured with NvMWriteBlockOnce == TRUE [ECUC_NvM_00072], the NvM module shall only write once to the associated NV memory, i.e in case of a blank NV device.⌋*(SRS_Mem_08015)*

[SWS_NvM_00398] ⌈For NVRAM blocks configured with NvMWriteBlockOnce == TRUE, the NvM module shall not allow disabling the write protection explicitly using the NvM_SetBlockProtection function.[SWS_NvM_00450]⌋*()*

[SWS_NvM_00952] ⌈For a block configured with NVM_WRITE_BLOCK_ONCE (TRUE), NvM shall reject any Write/Erase/Invalidate request made prior to the first read request.⌋*()*

Note: In case of a reset, the write protection flag of a block configured with NVM_WRITE_BLOCK_ONCE (TRUE), from the NvM Administrative block, is cleared. In order to reactivate the protection, the block has to be read prior to a first Write/Erase/Invalidate request being processed, in order to set the write proctection only for a block that is valid and consistent. The first read request can be done either as a single block request or as part of NvM_ReadAll.

7.2.2.14 Validation and modification of RAM block data

This chapter shall give summarized information regarding the internal handling of NVRAM Manager status bits. Depending on different API calls, the influence on the status of RAM blocks shall be described in addition to the specification items located in chapter 8.3.1. The following figures depict the state transitions of RAM blocks.

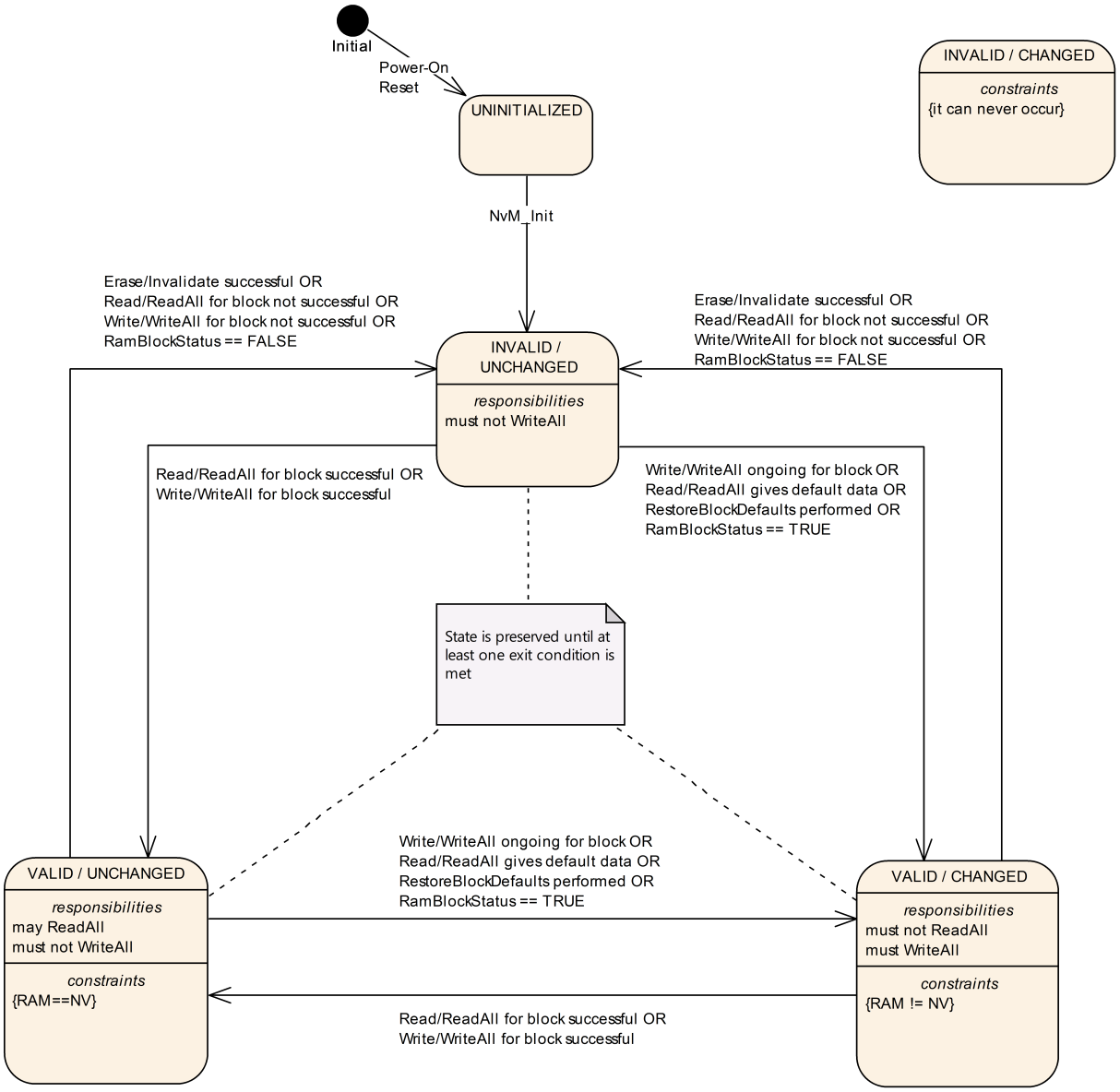

Figure 7.8: RAM Block States

Since entering and preserving a state can be done based on multiple conditions and placing them all in the above figure would make it difficult to understand, more detailed explanations are provided in the following subchapters. The INVALID / CHANGED state is not detailed as it can never be reached (as mentioned in the figure above).

After the Initialization the RAM Block is in state INVALID/UNCHANGED until it is updated via NvM_ReadAll, which causes a transition to state VALID/UNCHANGED. In this state WriteAll is not allowed. This state is left, if the NvM_SetRamBlockStatus is invoked. If there occurs a CRC error the RAM Block changes to state INVALID again, which than can be left via the implicit or explicit error recovery mechanisms. After error recovery the block is in state VALID/CHANGED as the content of the RAM differs from the NVRAM content.

[SWS_NvM_00344] ⌈If the API for modifying the RAM block status has been disabled in configuration (via NvMSetRamBlockStatusApi or NvMBlockUseSetRamBlockStatus) the NvM module shall treat a RAM block or the RAM mirror in the NvM module (in case of explicit synchronization) as valid and changed when writing data in the corresponding NV block, i.e. during NvM_WriteAll, the NvM module shall write each permanent RAM block to NV memory.⌋*(SRS_Mem_08550)*

[SWS_NvM_00345] ⌈If the API for modifying the RAM block status has been disabled in configuration (via NvMSetRamBlockStatusApi or NvMBlockUseSetRamBlockStatus) the NvM module shall treat a RAM block as invalid when reading data from NV block, i.e. during NvM_ReadAll, the NvM module shall copy each NVRAM block to RAM if configured accordingly.⌋*(SRS_Mem_08550)*

[SWS_NvM_00696] ⌈In case of an unsuccessful block read attempt, it is the responsibility of the application to provide valid data before the next write attempt.⌋*(SRS_Mem_08550)*

[SWS_NvM_00472] ⌈In case a RAM block is successfully copied to NV memory the RAM block state shall be set to "valid/unmodified" afterwards.⌋*(SRS_Mem_08541)*

7.2.2.14.1 The VALID / UNCHANGED state

This state implies that the contents of the RAM Block are either identical to the contents of the corresponding NV Block or - if the application has accessed the RAM Block - a potential change was not yet indicated. For a DATASET block these conditions apply to he RAM contents of the instance that was last processed. Also, the last block operation was successful and the block was not invalidated by request.

To enter the VALID / UNCHANGED state, at least of the following has to occur:

1. NvM_ReadAll() read successfully the block
2. NvM_ReadBlock finished successfully for the block
3. NvM_WriteBlock finished successfully for the block
4. NvM_WriteAll() wrote successfully the block

The VALID / UNCHANGED state is preserved while:

- the last read or write for a BlockID was successful (no error and no retrieval of default data)

AND

- the application has not indicated a potential change of RAM block since last read or write

7.2.2.14.2 The VALID / CHANGED state

This state implies that the contents of the RAM Block potentially differ from the contents of the corresponding NV Block. For a DATASET block this condition applies to the RAM contents of the instance that was last processed. Also, the last operation for the block was successsful and the block was not invalidated by request. The block owner can signal a potential RAM contents changed for the block causing the block state to become VALID / CHANGED.

To enter the VALID / CHANGED state, at least one of the following has to occur:

1. NvM_SetRamBlockStatus called with TRUE for the block
2. NvM_WriteBlock is called for the block
3. NvM_WriteAll will also process the block
4. NvM_ReadBlock called for the block gives default data
5. NvM_RestoreBlockDefaults called for the block finishes successfully
6. NvM_ReadAll gives default data when processign the block
7. NvM_ValidateAll processed successfully the block

The VALID / CHANGED state is preserved while:

- a block owner has indicated a potential change of RAM block

OR

- default data was retrieved (implicitly or explicitly) for the block upon last read

7.2.2.14.3 The INVALID / UNCHANGED state

This state implies that the NV Block is invalid. For a DATASET block this means that the NV Block contents are invalid for the last instance that was processed.

To enter the INVALID / UNCHANGED state, at least one of the following has to occur:

1. NvM_SetRamBlockStatus called with FALSE for the block
2. NvM_ReadBlock indicates invalidation by user request for the block
3. NvM_ReadBlock indicates corrupted data (if CRC configured) for the block
4. NvM_ReadBlock indicates wrong StaticID (if configured) for the block
5. NvM_WriteBlock finished non-successfully for the block
6. NvM_WriteAll non-successful write for the block
7. NvM_InvalidateNvBlock finished successfully for the block

8. NvM_EraseNvBlock finished successfully for the block

The INVALID / UNCHANGED state is preserved while:

- the block state is unknown at the time (early init, until ReadAll or first operation requested for a given block)

OR

- the block was detected as corrupted or with wrong StaticID

OR

- the last successful operation on the block was an invalidation

OR

- the current read failed and no default data

OR

- the last successful operation on the block was an erase

7.2.2.15 Communication and implicit synchronization between application and NVRAM manager

To minimize locking/unlocking overhead or the use of other synchronization methods, the communication between applications and the NvM module has to follow a strict sequence of steps which is described below. This ensures a reliable communication between applications and the NvM module and avoids data corruption in RAM blocks and a proper synchronization is guaranteed.

This access model assumes that two parties are involved in communication with a RAM block: The application and the NvM module.

[SWS_NvM_00697] ⌈If several applications are using the same RAM block it is not the job of the NvM module to ensure the data integrity of the RAM block. In this case, the applications have to synchronize their accesses to the RAM block and have to guarantee that no unsuitable accesses to the RAM block take place during NVRAM operations (details see below). Especially if several applications are sharing a NVRAM block by using (different) temporary RAM blocks, synchronization between applications becomes more complex and this is not handled by the NvM module, too. In case of using callbacks as notification method, it could happen that e.g. an application gets a notification although the request has not been initiated by this application.

All applications have to adhere to the following rules.⌋*()*

7.2.2.15.1 Write requests (NvM_WriteBlock or NvM_WritePRAMBlock)

[SWS_NvM_00698] ⌈Applications have to adhere to the following rules during write request for implicit synchronization between application and NVRAM manager:

1. The application fills a RAM block with the data that has to be written by the NvM module
2. The application issues the NvM_WriteBlock or NvM_WritePRAMBlock request which transfers control to the NvM module.
3. From now on the application should not modify the RAM block until success or failure of the request is signaled or derived via polling. In the meantime the contents of the RAM block may be read.
4. An application can use polling to get the status of the request or can be informed via a callback function asynchronously.
5. After completion of the NvM module operation, the RAM block is reusable for modifications.

⌋(*SRS_Mem_00013*)

7.2.2.15.2 Read requests (NvM_ReadBlock or NvM_ReadPRAMBlock)

[SWS_NvM_00699] ⌈Applications have to adhere to the following rules during read request for implicit synchronization between application and NVRAM manager:

1. The application provides a RAM block that has to be filled with NVRAM data from the NvM module's side.
2. The application issues the NvM_ReadBlock request which transfers control to the NvM module.
3. From now on the application should not read or write to the RAM block until success or failure of the request is signaled or derived via polling.
4. An application can use polling to get the status of the request or can be informed via a callback function.
5. After completion of the NvM module operation, the RAM block is available with new data for use by the application.

⌋(*SRS_Mem_00013*)

7.2.2.15.3 Restore default requests (NvM_RestoreBlockDefaults and NvM_RestorePRAMBlockDefaults)

[SWS_NvM_00700] ⌈Applications have to adhere to the following rules during restore default requests for implicit synchronization between application and NVRAM manager:

1. The application provides a RAM block, which has to be filled with ROM data from the NvM modules side.
2. The application issues the NvM_RestoreBlockDefaults or NvM_RestorePRAMBlockDefaults request which transfers control to the NvM module.
3. From now on the application should not read or write to the RAM block until success or failure of the request is signaled or derived via polling.
4. An application can use polling to get the status of the request or can be informed via a callback function.
5. After completion of the NvM module operation, the RAM block is available with the ROM data for use by the application.

⌋*(SRS_Mem_08548)*

7.2.2.15.4 Multi block read requests (NvM_ReadAll)

This request may be triggered only by the BSW Mode Manager at system startup.

This request fills all configured permanent RAM blocks with necessary data for startup.

If the request fails or the request is handled only partially successful, the NVRAMManager signals this condition to the DEM and returns an error to the BSW Mode Manager. The DEM and the BSW Mode Manager have to decide about further measures that have to be taken. These steps are beyond the scope of the NvM module and are handled in the specifications of DEM and BSW Mode Manager.

[SWS_NvM_00701] ⌈Applications have to adhere to the following rules during multi block read requests for implicit synchronization between application and NVRAM manager:

The BSW Mode Manager issues the NvM_ReadAll.

1. The BSW Mode Manager can use polling to get the status of the request or can be informed via a callback function.
2. During NvM_ReadAll, a single block callback (if configured) will be invoked after having completely processed a NVRAM block. These callbacks enable the RTE to start each SW-C individually.

⌋*()*

7.2.2.15.5 Multi block write requests (NvM_WriteAll)

This request should only be triggered by the BSW Mode Manager at shutdown of the system. This request writes the contents of all modified permanent RAM blocks to NV memory. By calling this request only during ECU shutdown, the BSW Mode Manager can ensure that no SW component is able to modify data in the RAM blocks until the end of the operation. These measures are beyond the scope of the NvM module and are handled in the specifications of the BSW Mode Manager.

[SWS_NvM_00702] ⌈Applications have to adhere to the following rules during multi block write requests for implicit synchronization between application and NVRAM manager:

1. The BSW Mode Manager issues the NvM_WriteAll request which transfers control to the NvM module.
2. The BSW Mode Manager can use polling to get the status of the request or can be informed via a callback function.

⌋*()*

7.2.2.15.6 Cancel Operation (NvM_CancelWriteAll)

This request cancels a pending NvM_WriteAll request. This is an asynchronous request and can be called to terminate a pending NvM_WriteAll request.

[SWS_NvM_00703] ⌈NvM_CancelWriteAll request shall only be used by the BSW Mode Manager.⌋*()*

7.2.2.15.7 Modification of administrative blocks

For administrative purposes an administrative block is part of each configured NVRAM block (ref. to ch. 7.1.3.4).

[SWS_NvM_00704] ⌈If there is a pending single-block operation for a NVRAM block, the application is not allowed to call any operation that modifies the administrative block, like NvM_SetDataIndex, NvM_SetBlockProtection, NvM_SetRamBlockStatus, until the pending job has finished.⌋*()*

7.2.2.16 Normal and extended runtime preparation of NVRAM blocks

This subchapter is supposed to provide a short summary of normal and extended runtime preparation of NVRAM blocks. The detailed behavior regarding the handling of NVRAM blocks during start-up is specified in chapter 8.3.3.1.

Depending on the two configuration parameters NvMDynamicConfiguration and NvMResistantToChangedSw the NVRAM Manager shall behave in different ways during start-up, i.e. while processing the request NvM_ReadAll().

If NvMDynamicConfiguration is set to FALSE, the NVRAM Manager shall ignore the stored configuration ID (see SWS_NvM_00034) and continue with the normal runtime preparation of NVRAM blocks. In this case the RAM block shall be checked for its validity. If the RAM block content is detected to be invalid the NV block shall be checked for its validity. A NV block which is detected to be valid shall be copied to its assigned RAM block. If an invalid NV Block is detected default data shall be loaded.

If NvMDynamicConfiguration is set to TRUE and a configuration ID mismatch is detected, the extended runtime preparation shall be performed for those NVRAM blocks which are configured with NvMResistantToChangedSw(FALSE). In this case default data shall be loaded independent of the validity of an assigned RAM or NV block.

7.2.2.17 Communication and explicit synchronization between application and NVRAM manager

In contrast to the implicit synchronization between the application and the NvM module (see section 7.2.2.15) an optional (i.e. configurable) explicit synchronization mechanism is available. It is realized by a RAM mirror in the NvM module. The data is transferred by the application in both directions via callback routines, called by the NvM module.

Here is a short analysis of this mechanism:

The advantage is that applications can control their data in a better way. They are responsible for copying consistent data to and from the NvM module's RAM mirror, so they know the point in time. The RAM block is never in an inconsistent state due to concurrent accesses.

The drawbacks are the additional RAM which needs to have the same size as the largest NVRAM block that uses this mechanism and the necessity of an additional copy between two RAM locations for every operation.

This mechanism especially enables the sharing of NVRAM blocks by different applications, if there is a module that synchronizes these applications and is the owner of the NVRAM block from the NvM module's perspective.

[SWS_NvM_00511] ⌈For every NVRAM block there shall be the possibility to configure the usage of an explicit synchronization mechanism by the parameter NvMBlockUseSyncMechanism.⌋*()*

[SWS_NvM_00512] ⌈The NvM module shall not allocate a RAM mirror if no block is configured to use the explicit synchronization mechanism.⌋*()*

[SWS_NvM_00513] ⌈The NvM module shall allocate only one RAM mirror if at least one block is configured to use the explicit synchronization mechanism. This RAM mirror

should not exceed the size of the longest NVRAM block configured to use the explicit synchronization mechanism.⌋()

[SWS_NvM_00514] ⌈The NvM module shall use the internal mirror as buffer for all operations that read and write the RAM block of those NVRAM blocks with NvMBlock UseSyncMechanism == TRUE. The buffer should not be used for the other NVRAM blocks.⌋()

[SWS_NvM_00515] ⌈The NvM module shall call the routine NvMWriteRamBlockToNv Callback in order to copy the data from the RAM block to the mirror for all NVRAM blocks with NvMBlockUseSyncMechanism == TRUE. This routine shall not be used for the other NVRAM blocks.⌋()

[SWS_NvM_00516] ⌈The NvM module shall call the routine NvMReadRamBlockFrom NvCallback in order to copy the data from the mirror to the RAM block for all NVRAM blocks with NvMBlockUseSyncMechanism == TRUE. This routine shall not be used for the other NVRAM blocks.⌋()

[SWS_NvM_00517] ⌈During a single block request if the routines NvMReadRamBlock FromNvCallback return E_NOT_OK, then the NvM module shall retry the routine call NvMRepeatMirrorOperations times. Thereafter the single block read job shall set the block specific request result to NVM_REQ_NOT_OK and shall report NVM_E_REQ_ FAILED to the DEM.⌋()

[SWS_NvM_00839] ⌈In the case the NvMReadRamBlockFromNvCallback routine returns E_NOT_OK, the NvM module shall retry the routine call in the next call of the Nv M_MainFunction.⌋()

[SWS_NvM_00579] ⌈During a single block request if the routines NvMWriteRamBlock ToNvCallback return E_NOT_OK, then the NvM module shall retry the routine call Nv MRepeatMirrorOperations times. Thereafter the single block write job shall set the block specific request result to NVM_REQ_NOT_OK and shall report NVM_E_REQ_ FAILED to the DEM.⌋()

[SWS_NvM_00840] ⌈In the case the NvMWriteRamBlockToNvCallback routine returns E_NOT_OK, the NvM module shall retry the routine call in the next call of the NvM_ MainFunction.⌋()

[SWS_NvM_00837] ⌈During a multi block request (NvM_WriteAll) if the routines Nv MWriteRamBlockToNvCallback return E_NOT_OK, then the NvM module shall retry the routine call NvMRepeatMirrorOperations times. Thereafter the job of the function NvM_WriteAll shall set the block specific request result to NVM_REQ_NOT_OK and shall report NVM_E_REQ_FAILED to the DEM.⌋()

[SWS_NvM_00838] ⌈During a multi block request (NvM_ReadAll) if the routines Nv MReadRamBlockFromNvCallback return E_NOT_OK, then the NvM module shall retry the routine call NvMRepeatMirrorOperations times. Thereafter the job of the function NvM_ReadAll shall set the block specific request result to NVM_REQ_NOT_OK and shall report NVM_E_REQ_FAILED to the DEM.⌋()

[SWS_NvM_00904] ⌈If a block has explicit synchronization configured for it then it shall not have a permanent RAM image configured.⌋*(SRS_Mem_08534)*

The following two sections clarify the differences when using the explicit synchronization mechanism, compare to 7.2.2.17.1 and 7.2.2.17.2.

7.2.2.17.1 Write requests (NvM_WriteBlock or NvM_WritePRAMBlock)

[SWS_NvM_00705] ⌈Applications have to adhere to the following rules during write request for explicit synchronization between application and NVRAM manager:

1. The application fills a RAM block with the data that has to be written by the NvM module.
2. The application issues the NvM_WriteBlock or NvM_WritePRAMBlock request.
3. The application might modify the RAM block until the routine NvMWriteRamBlock ToNvCallback is called by the NvM module.
4. If the routine NvMWriteRamBlockToNvCallback is called by the NvM module, then the application has to provide a consistent copy of the RAM block to the destination requested by the NvM module.

   The application can use the return value E_NOT_OK in order to signal that data was not consistent. The NvM module will accept this NvMRepeatMirrorOperations times and then postpones the request and continues with its next request.
5. Continuation only if data was copied to the NvM module:
6. From now on the application can read and write the RAM block again.
7. An application can use polling to get the status of the request or can be informed via a callback routine asynchronously.

Note: The application may combine several write requests to different positions in one RAM block, if NvM_WriteBlock or NvM_WritePRAMBlock was requested, but not yet processed by the NvM module. The request was not processed, if the callback routine NvMWriteRamBlockToNvCallback was not called.⌋*()*

7.2.2.17.2 Read requests (NvM_ReadBlock or NvM_ReadPRAMBlock)

[SWS_NvM_00706] ⌈Applications have to adhere to the following rules during read request for explicit synchronization between application and NVRAM manager:

1. The application provides a RAM block that has to be filled with NVRAM data from the NvM module's side.
2. The application issues the NvM_ReadBlock or NvM_ReadPRAMBlock request.

3. The application might modify the RAM block until the routine NvMReadRamBlock FromNvCallback is called by the NvM module.
4. If the routine NvMReadRamBlockFromNvCallback is called by the NvM module, then the application copy the data from the destination given by the NvM module to the RAM block.The application can use the return value E_NOT_OK in order to signal that data was not copied. The NvM module will accept this NvMRepeat MirrorOperations times and then postpones the request and continues with its next request.
5. Continuation only if data was copied from the NvM module:
6. Now the application finds the NV block values in the RAM block.
7. The application can use polling to get the status of the request or can be informed via a callback routine.

Note: The application may combine several read requests to different positions in one NV block, if NvM_ReadBlock or NvM_ReadPRAMBlock was requested, but not yet processed by the NvM module. The request was not processed, if the callback routine NvMReadRamBlockFromNvCallback was not called.

Note: NvM_RestoreBlockDefaults and NvM_RestorePRAMBlockDefaults works similarly to NvM_ReadBlock.⌋*()*

7.2.2.17.3 Multi block read requests (NvM_ReadAll)

This request may be triggered only by the BSW Mode Manager at system startup. This request fills all configured permanent RAM blocks with necessary data for startup.

If the request fails or the request is handled only partially successful, the NVRAM-Manager signals this condition to the DEM and returns an error to the BSW Mode Manager. The DEM and the BSW Mode Manager have to decide about further measures that have to be taken. These steps are beyond the scope of the NvM module and are handled in the specifications of DEM and BSW Mode Manager.

Normal operation:

1. The BSW Mode Manager issues the NvM_ReadAll.
2. The BSW Mode Manager can use polling to get the status of the request or can be informed via a callback function.
3. During NvM_ReadAll job, if a synchronization callback (NvM_ReadRamBlock FromNvm) is configured for a block it will be called by the NvM module. In this callback the application shall copy the data from the destination given by the Nv M module to the RAM block.The application can use the return value E_NOT_OK in order to signal that data was not copied. The NvM module will accept this Nv MRepeatMirrorOperations times and then report the read operation as failed.

4. Now the application finds the NV block values in the RAM block if the read operation was successful.
5. During NvM_ReadAll, a single block callback (if configured) will be invoked after having completely processed a NVRAM block. These callbacks enable the RTE to start each SW-C individually.
6. After processing of the last block and calling its single block callback (if configured), the multi block callback (if configured) will be invoked.

7.2.2.17.4 Multi block write requests (NvM_WriteAll)

This request should only be triggered by the BSW Mode Manager at shutdown of the system. This request writes the contents of all modified permanent RAM blocks to NV memory. By calling this request only during ECU shutdown, the BSW Mode Manager can ensure that no SW component is able to modify data in the RAM blocks until the end of the operation. These measures are beyond the scope of the NvM module and are handled in the specifications of the BSW Mode Manager.

Normal operation:

1. The BSW Mode Manager issues the NvM_WriteAll request which transfers control to the NvM module.
2. During NvM_WriteAll job, if a synchronization callback (NvM_WriteRamBlockTo NvM) is configured for a block it will be called by the NvM module. In this callback the application has to provide a consistent copy of the RAM block to the destination requested by the NvM module.

   The application can use the return value E_NOT_OK in order to signal that data was not consistent. The NvM module will accept this NvMRepeatMirrorOperations times and then report the write operation as failed.
3. Now the application can read and write the RAM block again.
4. The BSW Mode Manager can use polling to get the status of the request or can be informed via a callback function.

7.2.2.18 Static Block ID Check

Note: NVRAM Manager stores the NV Block Header including the Static Block ID in the NV Block each time the block is written to NV memory. When a block is read, its Static Block ID is compared to the requested block ID. This permits to detect hardware failures which cause a wrong block to be read.

[SWS_NvM_00523] ⌈The NVRAM Manager shall store the Static Block ID field of the Block Header each time the block is written to NV memory.⌋ *(SRS_Mem_08555)*

[SWS_NvM_00524] ⌈The NVRAM Manager shall check the Block Header each time the block is read from NV memory.⌋ *(SRS_Mem_08555)*

[SWS_NvM_00525] ⌈If the Static Block ID check fails then the failure NVM_E_WRONG_BLOCK_ID is reported to DEM.⌋ *()*

[SWS_NvM_00580] ⌈If the Static Block ID check fails then the read error recovery is initiated. Hint: A check shall be made during configuration to ensure that all Static Block IDs are unique.⌋ *()*

7.2.2.19 Read Retry

[SWS_NvM_00526] ⌈If the NVRAM manager detects a failure during a read operation from NV memory, a CRC error then one or more additional read attempts shall be made, as configured by NVM_MAX_NUM_OF_READ_RETRIES, before continuing to read the redundant NV Block.⌋ *(SRS_Mem_08554)*

[SWS_NvM_00581] ⌈If the NVRAM manager detects a failure during a read operation from NV memory, a CRC error then one or more additional read attempts shall be made, as configured by NVM_MAX_NUM_OF_READ_RETRIES, before continuing to read the ROM Block.⌋ *(SRS_Mem_08554)*

[SWS_NvM_00582] ⌈If the NVRAM manager detects a failure during a read operation from NV memory, a Static Block ID check then one or more additional read attempts shall be made, as configured by NVM_MAX_NUM_OF_READ_RETRIES, before continuing to read the redundant NV Block.⌋ *(SRS_Mem_00129)*

[SWS_NvM_00583] ⌈If the NVRAM manager detects a failure during a read operation from NV memory, a Static Block ID check then one or more additional read attempts shall be made, as configured by NVM_MAX_NUM_OF_READ_RETRIES, before continuing to read the ROM Block.⌋ *()*

7.2.2.20 Write Verification

When a RAM Block is written to NV memory the NV block shall be immediately read back and compared with the original content in RAM Block if the behaviour is enabled by NVM_WRITE_VERIFICATION.

[SWS_NvM_00527] ⌈Comparison between original content in RAM Block and the block read back shall be performed in steps so that the number of bytes read and compared is not greater than as specified by the configuration parameter NVM_WRITE_VERIFICATION_DATA_SIZE.⌋ *(SRS_Mem_08554, SRS_Mem_08556)*

[SWS_NvM_00528] ⌈If the original content in RAM Block is not the same as read back then the production code error NVM_E_VERIFY_FAILED shall be reported to DEM.⌋ *(SRS_Mem_08556)*

[SWS_NvM_00529] ⌈If the original content in RAM Block is not the same as read back then write retries shall be performed as specified in this document.⌋*(SRS_Mem_-08554, SRS_Mem_08556)*

[SWS_NvM_00530] ⌈If the read back operation fails then no read retries shall be performed.⌋*()*

[SWS_NvM_00897] ⌈If the original content in RAM Block is not the same as read back, for the initial write attempt as well as for all the configured retries, then NvM shall set as request result NVM_REQ_NOT_OK.⌋*(SRS_Mem_00017, SRS_Mem_08554, SWS_NvM_08541, SRS_Mem_00030, SRS_Mem_08556)*

7.2.2.21 Comparing NV data in NvM

In order to avoid unnecessary write operations in NV memory, if the NV data of a specific RAM Block was not updated during runtime, the NvM module offers a CRC based compare mechanism which can be applied while processing a write job.

[SWS_NvM_00849] ⌈The NvM module shall provide an option to skip writing of unchanged data by implementing a CRC based compare mechanism.⌋*(SRS_Mem_-00136)*

Note: In general, there is a risk that some changed content of an RAM Block leads to the same CRC as the initial content so that an update might be lost if this option is used. Therefore this option should be used only for blocks where this risk can be tolerated.

[SWS_NvM_00850] ⌈For every NVRAM Block there shall be the possibility to configure the usage of the CRC based compare mechanism by the parameter NvMBlockUse CRCCompMechanism if the parameter NvMBlockUseCrc is set to true.⌋*(SRS_Mem_-00136)*

7.2.2.22 NvM and BswM interaction

[SWS_NvM_00745] ⌈The NvM shall use the BswM API BswM_NvM_CurrentJob Mode() when it needs to inform the BswM about a multiblock request state change.⌋*()*

[SWS_NvM_00950] ⌈If NvMBswMMultiBlockJobStatusInformation is true, the NvM shall not call the configured multiblock callback.⌋*()*

[SWS_NvM_00746] ⌈The NvM shall use the BswM API BswM_NvM_CurrentBlock Mode() when it needs to inform the BswM about a single block request acceptance (as being pending) and result.⌋*()*

[SWS_NvM_00888] ⌈If NvMBswMMultiBlockJobStatusInformation is true, when NvM accepts a multiblock operation the NvM shall inform the BswM about the accepted multiblock operation as being pending, by calling the BswM_NvM_CurrentJobMode

with the related multiblock request type and, as mode, NVM_REQ_PENDING.⌋*(SRS_MEM_00020, SRS_MEM_00136, SRS_MEM_08535, SRS_MEM_08533)*

[SWS_NvM_00889] ⌈If NvMBswMMultiBlockJobStatusInformation is true, when a multiblock operation finishes or is canceled the NvM shall inform the BswM about the result of the multiblock operation, by calling the BswM_NvM_CurrentJobMode with the related multiblock request type and, as mode, the outcome of the multiblock operation.⌋ *(SRS_MEM_00020, SRS_MEM_00136, SRS_MEM_08535, SRS_MEM_08533)*

[SWS_NvM_00890] ⌈If NvMBswMBlockStatusInformation is true, when NvM accepts a single block operation the NvM shall inform the BswM about the accepted single block operation as being pending, by calling the BswM_NvM_CurrentBlockMode with the related Block ID and, as mode, NVM_REQ_PENDING.⌋*(SRS_MEM_00125, SRS_MEM_00020)*

[SWS_NvM_00891] ⌈If NvMBswMBlockStatusInformation is true, when a single block operation finishes or is canceled the NvM shall inform the BswM about the result of the single block operation, by calling the BswM_NvM_CurrentBlockMode with the related Block ID and, as mode, the outcome of the singleblock operation.⌋*(SRS_MEM_00125, SRS_MEM_00020)*

[SWS_NvM_00892] ⌈If NvMBswMBlockStatusInformation is true and NvM has a multiblock operation ongoing, for each block processed due to the multiblock operation, NvM shall inform the BswM when it starts to process the block, as being pending, by calling the BswM_NvM_CurrentBlockMode with the related Block ID and, as mode, NVM_REQ_PENDING.⌋*(SRS_MEM_00125, SRS_MEM_00020)*

[SWS_NvM_00949] ⌈If NvMBswMBlockStatusInformation is true and NvM has a multiblock operation ongoing, for each block processed due to the multiblock operation, NvM shall inform the BswM about the result of the processing of the block when the block is finished processing, by calling the BswM_NvM_CurrentBlockMode with the related Block ID and, as mode, the outcome of the singleblock operation.⌋*(SRS_MEM_00125, SRS_MEM_00020)*

7.2.2.23 NvM behaviour in case of Block locked

The NvM_SetBlockLockStatus API service shall only be usable by BSW Components, it is not published as Service in the SWC-Description. Thus it will not be accessible via RTE.

[SWS_NvM_00751] ⌈If the function NvM_SetBlockLockStatus was called with the parameter BlockLocked as TRUE, the NvM shall guarantee that the NV contents associated to the NVRAM block identified by BlockId, will not be modified by any request. The Block shall be skipped during NvM_WriteAll, other requests, that are NvM_WriteBlock, NvM_WritePRAMBlock, NvM_InvalidateNvBlock, NvM_EraseNvBlock, shall be rejected.⌋*()*

[SWS_NvM_00752] ⌈If the function NvM_SetBlockLockStatus was called with the parameter BlockLocked as TRUE, the NvM shall guarantee that at next start-up, during processing of NvM_ReadBlock or NvM_ReadPRAMBlock, this NVRAM block shall be loaded from NV memory.⌋*()*

[SWS_NvM_00753] ⌈If the function NvM_SetBlockLockStatus was called with the parameter BlockLocked as FALSE, the NvM shall guarantee normal processing of this NVRAM block as specified by AUTOSAR.⌋*()*

[SWS_NvM_00754] ⌈The BlockLocked setting made using the function NvM_SetBlock LockStatus shall not be changeable by NvM_SetRamBlockStatus, nor by NvM_Set BlockProtection.⌋*()*

7.2.2.23.1 Use Case

Save new Data for an NVRAM block via diagnostic services into NV memory. These data shall be made available to the SW-C(s) with next ECU start-up, i.e. they shall neither be overwritten by a request originating from an SW-C, nor be overwritten with permanent RAM block's data during shut-down (NvM_WriteAll).

7.2.2.23.2 Usage (by DCM):

1. DCM requests NvM_SetBlockLockStatus(<BlockId>, FALSE), in order to re-enable writing to this block. (It might be locked by executing this procedure before).
2. DCM requests NvM_WriteBlock(<blockId>, <DataBuffer>)
3. DCM polls for completion of write request (using NvM_GetErrorStatus())
4. On success (NVM_REQ_OK), the DCM issues NvM_SetBlockLockStatus(<Block Id>, TRUE).

7.2.2.24 Block Compression

The block data is compressed before it is written to NV memory. The type of compression (block split, compression, delta) is vendor-specific.

The use-case is for larger data blocks with changes of only smaller junks (like drive-cycle logging). The goal is that not the whole block needs to be written to NV memory to reduce the overall write-cycles.

The block split would divide the block in multiple sub-blocks and only the changed sub-blocks would be written. Alternatively, only the changed delta could be written. Anyway, any data compression algorithm could be used.

The drawback is always a higher runtime for writing or reading the data.

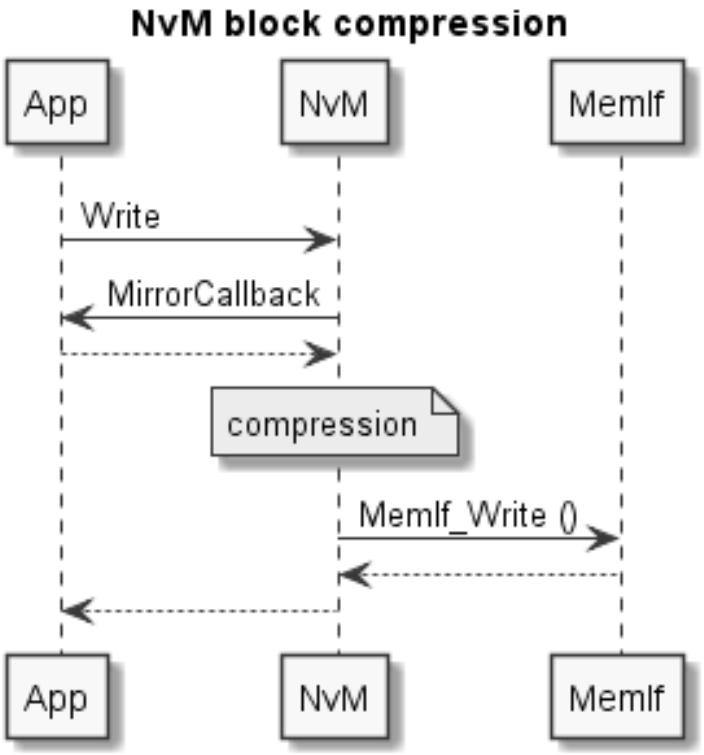

Figure 7.9: NvM block compression

[SWS_NvM_00966] ⌈DRAFT In case the NvMBlockUseCompression is set to true, the NvM shall compress the stored data in NV memory.⌋ *()*

7.2.2.25 Block Ciphering

For security purposes NvM supports synchronous encryption and decryption via CSM module using symmetric 16 byte aligned algorithms, e.g. AES128.

The user always works with plain data, the NV RAM stores the ciphered data:

> Write data: NvM encrypts the plain user data and then forwards the ciphered data to the device.

> Read data: NvM reads the ciphered data from device, decrypts the data and finally provides the plain data to the user.

To check the integrity of the ciphered data a CRC can be configured (as usual). NvM will then calculate the CRC over encrypted data and recalculate and check the CRC before decryption: the CRC always matches the ciphered data.

[SWS_NvM_00976] ⌈In case NvMBlockCipheringRef is given, the NvM shall before forwarding the write request to MemIf encrypt the plain data using Csm_Encrypt() with the CSM job given in NvMCsmEncryptionJobReference.

The CRC calculation (if configured) shall be done over the encrypted data.⌋*()*

[SWS_NvM_00977] ⌈In case Csm_Encrypt() returns a CRYPTO_E_BUSY, the NvM shall retry to redo the job. After NvMCsmRetryCounter times of retry the NvM shall abort the write job and set the NvM result to NVM_REQ_NOT_OK and signal an error via NvM_JobErrorNotification().⌋*()*

[SWS_NvM_00978] ⌈In case Csm_Encrypt() returns any other error than CRYPTO_E_BUSY or CRYPTO_E_OK, the NvM shall abort the write job and set the NvM result to NVM_REQ_NOT_OK and signal an error via NvM_JobErrorNotification().⌋*()*

[SWS_NvM_00979] ⌈In case Csm_Encrypt() returns successfully with CRYPTO_E_OK, the NvM shall continue the write job (e.g. with the CRC calculation) with the new length given in NvMNvBlockNVRAMDataLength.

In case of the returned length in resultLengthPtr is different to the NvMNvBlock NVRAMDataLength the development error NVM_E_BLOCK_CHIPHER_LENGTH_MISSMATCH shall be triggerd.⌋*()*

[SWS_NvM_00980] ⌈In case NvMBlockCipheringRef is given, the NvM shall before forwarding the read request to application decrypt the stored data using Csm_Decrypt() with the CSM job given in NvMCsmDecryptionJobReference. The CRC check (if configured) shall be done over the encrypted data. If the CRC does not match, NvM will not decrypt the data but abort the job with NVM_REQ_INTEGRITY_FAILED.⌋*()*

[SWS_NvM_00981] ⌈In case Csm_Decrypt() returns a CRYPTO_E_BUSY, the NvM shall retry to redo the job. After NvMCsmRetryCounter times of retry the NvM shall abort the read job and set the NvM result to NVM_REQ_NOT_OK and signal an error via NvM_JobErrorNotification().⌋*()*

[SWS_NvM_00982] ⌈In case Csm_Decrypt() returns any other error than CRYPTO_E_BUSY or CRYPTO_E_OK, the NvM shall abort the read job and set the NvM result to NVM_REQ_NOT_OK and signal an error via NvM_JobErrorNotification().⌋*()*

[SWS_NvM_00983] ⌈In case Csm_Decrypt() returns successfully with CRYPTO_E_OK, the NvM shall continue the read job with the new length given in NvMNvBlock Length.

In case of the returned length in resultLengthPtr is different to the NvMNvBlockLength the development error NVM_E_BLOCK_CHIPHER_LENGTH_MISSMATCH shall be triggerd.⌋*()*

7.3 Error Classification

7.3.1 Development Errors

[SWS_NvM_91004] ⌈

Type of error	*Related error code*	*Error value*
API is called with wrong parameter block ID	NVM_E_PARAM_BLOCK_ID	0x0A
API is called with wrong parameter block data	NVM_E_PARAM_BLOCK_DATA_IDX	0x0C
API is called with wrong parameter address	NVM_E_PARAM_ADDRESS	0x0D
API is called with wrong parameter data	NVM_E_PARAM_DATA	0x0E
API is called with wrong parameter pointer	NVM_E_PARAM_POINTER	0x0F
API is called for a block without defaults when either the NvM_RestoreBlockDeafults or NvM_RestorePRAMBlockDefaults is called for a valid block ID that has no default data and no NvMInit BlockCallback configured for the block	NVM_E_BLOCK_WITHOUT_DEFAULTS	0x11
API is called when NVRAM manager is not initialized yet	NVM_E_UNINIT	0x14
read/write/control API is called for a block which is already listed or in progress	NVM_E_BLOCK_PENDING	0x15
Service is not possible with this block configuration	NVM_E_BLOCK_CONFIG	0x18
write API is called for a block which RAM block is locked	NVM_E_BLOCK_LOCKED	0x19
write/erase/invalidate API is called for a block with MVM_WRITE_BLOCK_ONCE (TRUE) prior to the first read request for that block	NVM_E_WRITE_ONCE_STATUS_UNKNOWN	0x1A
The length resulting from encryption or decription do not match with the given length in the configuration.	NVM_E_BLOCK_CHIPHER_LENGTH_MISSMATCH	0x1B

⌋*(SRS_BSW_00385, SRS_BSW_00386, SRS_BSW_00406, SRS_BSW_00337, SRS_BSW_00327, SRS_BSW_00331)* (SRS_BSW_00385, SRS_BSW_00386, SRS_BSW_00406, SRS_BSW_00337, SRS_BSW_00327)

[SWS_NvM_00961] ⌈The development error NVM_E_WRITE_PROTECTED (0x1B) shall be detectable by the NvM module when a write attempt to a NVRAM block with write protection (which write protection can be either configured or set by explicit request) occurs.⌋*()*

[SWS_NvM_00027] ⌈If development error detection is enabled for NvM module, the function NvM_SetDataIndex shall report the DET error NVM_E_UNINIT when NVM is not yet initialized.⌋*(SRS_BSW_00323, SRS_BSW_00385, SRS_BSW_00386, SRS_BSW_00406, SRS_BSW_00327, SRS_BSW_00331)*

-

[SWS_NvM_00598] ⌈If development error detection is enabled for NvM module, the function NvM_SetDataIndex shall report the DET error NVM_E_BLOCK_PENDING when NVRAM block identifier is already queued or currently in progress.⌋*()*

-

[SWS_NvM_00599] ⌈If development error detection is enabled for NvM module, the function NvM_SetDataIndex shall report the DET error NVM_E_PARAM_BLOCK_DATA_IDX when DataIndex parameter exceeds the total number of configured datasets (Check: [SWS_NvM_00444], [SWS_NvM_00445]).⌋*()*

[SWS_NvM_00601] ⌈If development error detection is enabled for NvM module, the function NvM_SetDataIndex shall report the DET error NVM_E_PARAM_BLOCK_ID when the passed BlockID is out of range.⌋*()*

[SWS_NvM_00602] ⌈If development error detection is enabled for NvM module, the function NvM_GetDataIndex shall report the DET error NVM_E_UNINIT when NVM not yet initialized.⌋*()*

[SWS_NvM_00604] ⌈If development error detection is enabled for NvM module, the function NvM_GetDataIndex shall report the DET error NVM_E_PARAM_BLOCK_ID when the passed BlockID is out of range.⌋*()*

[SWS_NvM_00605] ⌈If development error detection is enabled for NvM module, the function NvM_GetDataIndex shall report the DET error NVM_E_PARAM_DATA when a NULL pointer is passed via the parameter DataIndexPtr.⌋*()*

[SWS_NvM_00606] ⌈If development error detection is enabled for NvM module, the function NvM_SetBlockProtection shall report the DET error NVM_E_UNINIT when NVM is not yet initialized.⌋*()*

[SWS_NvM_00607] ⌈If development error detection is enabled for NvM module, the function NvM_SetBlockProtection shall report the DET error NVM_E_BLOCK_PENDING when NVRAM block identifier is already queued or currently in progress.⌋*()*

[SWS_NvM_00608] ⌈If development error detection is enabled for NvM module, the function NvM_SetBlockProtection shall report the DET error NVM_E_BLOCK_CONFIG when the NVRAM block is configured with NvMWriteBlockOnce = TRUE.⌋*()*

[SWS_NvM_00609] ⌈If development error detection is enabled for NvM module, the function NvM_SetBlockProtection shall report the DET error NVM_E_PARAM_BLOCK_ID when the passed BlockID is out of range.⌋*()*

[SWS_NvM_00759] ⌈If development error detection is enabled for NvM module, the function NvM_SetBlockProtection shall report the DET error NVM_E_BLOCK_LOCKED when the block is locked.⌋*()*

[SWS_NvM_00610] ⌈If development error detection is enabled for NvM module, the function NvM_GetErrorStatus shall report the DET error NVM_E_UNINIT when NVM is not yet initialized.⌋*()*

[SWS_NvM_00611] ⌈If development error detection is enabled for NvM module, the function NvM_GetErrorStatus shall report the DET error NVM_E_PARAM_BLOCK_ID when the passed BlockID is out of range.⌋*()*

[SWS_NvM_00612] ⌈If development error detection is enabled for NvM module, the function NvM_GetErrorStatus shall report the DET error NVM_E_PARAM_DATA when a NULL pointer is passed via the parameter RequestResultPtr.⌋*()*

[SWS_NvM_00613] ⌈If development error detection is enabled for NvM module, the function NvM_GetVersionInfo shall report the DET error NVM_E_PARAM_POINTER when a NULL pointer is passed via the parameter versioninfo.⌋*()*

[SWS_NvM_00614] ⌈If development error detection is enabled for NvM module, the function NvM_ReadBlock shall report the DET error NVM_E_UNINIT when NVM is not yet initialized.⌋*()*

[SWS_NvM_00615] ⌈If development error detection is enabled for NvM module, the function NvM_ReadBlock shall report the DET error NVM_E_BLOCK_PENDING when NVRAM block identifier is already queued or currently in progress.⌋*()*

[SWS_NvM_00616] ⌈If development error detection is enabled for NvM module, the function NvM_ReadBlock shall report the DET error NVM_E_PARAM_ADDRESS when no permanent RAM block and no explicit synchronization are configured and a NULL pointer is passed via the parameter NvM_DstPtr.⌋*()*

[SWS_NvM_00618] ⌈If development error detection is enabled for NvM module, the function NvM_ReadBlock shall report the DET error NVM_E_PARAM_BLOCK_ID when the passed BlockID is out of range.⌋*()*

[SWS_NvM_00823] ⌈If development error detection is enabled for NvM module, the function NvM_ReadPRAMBlock shall report the DET error NVM_E_UNINIT when NVM is not yet initialized.⌋*()*

[SWS_NvM_00824] ⌈If development error detection is enabled for NvM module, the function NvM_ReadPRAMBlock shall report the DET error NVM_E_BLOCK_PENDING when NVRAM block identifier is already queued or currently in progress.⌋*()*

[SWS_NvM_00825] ⌈If development error detection is enabled for NvM module, the function NvM_ReadPRAMBlock shall report the DET error NVM_E_PARAM_ADDRESS when no permanent RAM block and no explicit synchronization are configured, for the received block ID.⌋*(SRS_Mem_00016, SRS_Mem_00038)*

[SWS_NvM_00826] ⌈If development error detection is enabled for NvM module, the function NvM_ReadPRAMBlock shall report the DET error NVM_E_PARAM_BLOCK_ID when the passed BlockID is out of range.⌋*()*

[SWS_NvM_00619] ⌈If development error detection is enabled for NvM module, the function NvM_WriteBlock shall report the DET error NVM_E_UNINIT when NVM not yet initialized.⌋*()*

[SWS_NvM_00620] ⌈If development error detection is enabled for NvM module, the function NvM_WriteBlock shall report the DET error NVM_E_BLOCK_PENDING when NVRAM block identifier is already queued or currently in progress.⌋*()*

[SWS_NvM_00622] ⌈If development error detection is enabled for NvM module, the function NvM_WriteBlock shall report the DET error NVM_E_PARAM_ADDRESS when no permanent RAM block and no explicit synchronization are configured and a NULL pointer is passed via the parameter NvM_SrcPtr.⌋*(SRS_Mem_00017, SRS_Mem_08541)*

[SWS_NvM_00624] ⌈If development error detection is enabled for NvM module, the function NvM_WriteBlock shall report the DET error NVM_E_PARAM_BLOCK_ID when the passed BlockID is out of range.⌋*()*

[SWS_NvM_00748] ⌈If development error detection is enabled for NvM module, the function NvM_WriteBlock shall report the DET error NVM_E_BLOCK_LOCKED when the block is locked.⌋*(SRS_Mem_08541, SRS_Mem_00127, SRS_Mem_00038)*

[SWS_NvM_00827] ⌈If development error detection is enabled for NvM module, the function NvM_WritePRAMBlock shall report the DET error NVM_E_UNINIT when NVM not yet initialized.⌋*()*

[SWS_NvM_00828] ⌈If development error detection is enabled for NvM module, the function NvM_WritePRAMBlock shall report the DET error NVM_E_BLOCK_PENDING when NVRAM block identifier is already queued or currently in progress.⌋*()*

[SWS_NvM_00893] ⌈If development error detection is enabled for NvM module, the function NvM_WritePRAMBlock shall report the DET error NVM_E_PARAM_ADDRESS when no permanent RAM block and no explicit synchronization are configured.⌋*(SRS_Mem_00018, SRS_Mem_08548)*

[SWS_NvM_00829] ⌈If development error detection is enabled for NvM module, the function NvM_WritePRAMBlock shall report the DET error NVM_E_PARAM_BLOCK_ID when the passed BlockID is out of range.⌋*()*

[SWS_NvM_00830] ⌈If development error detection is enabled for NvM module, the function NvM_WritePRAMBlock shall report the DET error NVM_E_BLOCK_LOCKED when the block is locked.⌋*()*

[SWS_NvM_00625] ⌈If development error detection is enabled for NvM module, the function NvM_RestoreBlockDefaults shall report the DET error NVM_E_UNINIT when NVM is not yet initialized.⌋*()*

[SWS_NvM_00626] ⌈If development error detection is enabled for NvM module, the function NvM_RestoreBlockDefaults shall report the DET error NVM_E_BLOCK_PENDING when NVRAM block identifier is already queued or currently in progress.⌋*()*

[SWS_NvM_00894] ⌈If development error detection is enabled for NvM module, the function NvM_RestorePRAMBlockDefaults shall report the DET error NVM_E_PARAM_ADDRESS when no permanent RAM block and no explicit synchronization are configured.⌋*(SRS_Mem_00018, SRS_Mem_08548)*

[SWS_NvM_00629] ⌈If development error detection is enabled for NvM module, the function NvM_RestoreBlockDefaults shall report the DET error NVM_E_PARAM_ADDRESS when no permanent RAM block and no explicit synchronization are configured and a NULL pointer is passed via the parameter NvM_DstPtr.⌋*(SRS_Mem_00016, SRS_Mem_08548)*

[SWS_NvM_00630] ⌈If development error detection is enabled for NvM module, the function NvM_RestoreBlockDefaults shall report the DET error NVM_E_PARAM_BLOCK_ID when the passed BlockID is out of range.⌋*()*

[SWS_NvM_00831] ⌈If development error detection is enabled for NvM module, the function NvM_RestorePRAMBlockDefaults shall report the DET error NVM_E_UNINIT when NVM is not yet initialized.⌋*()*

[SWS_NvM_00832] ⌈If development error detection is enabled for NvM module, the function NvM_RestorePRAMBlockDefaults shall report the DET error NVM_E_BLOCK_PENDING when NVRAM block identifier is already queued or currently in progress.⌋*()*

[SWS_NvM_00834] ⌈If development error detection is enabled for NvM module, the function NvM_RestorePRAMBlockDefaults shall report the DET error NVM_E_PARAM_BLOCK_ID when the passed BlockID is out of range.⌋*()*

[SWS_NvM_00631] ⌈If development error detection is enabled for NvM module, the function NvM_EraseNvBlock shall report the DET error NVM_E_UNINIT when the NVM is not yet initialized.⌋*()*

[SWS_NvM_00632] ⌈If development error detection is enabled for NvM module, the function NvM_EraseNvBlock shall report the DET error NVM_E_BLOCK_PENDING when the NVRAM block identifier is already queued or currently in progress.⌋*()*

[SWS_NvM_00635] ⌈If development error detection is enabled for NvM module, the function NvM_EraseNvBlock shall report the DET error NVM_E_PARAM_BLOCK_ID when the passed BlockID is out of range.⌋*()*

[SWS_NvM_00636] ⌈If development error detection is enabled for NvM module, the function NvM_EraseNvBlock shall report the DET error NVM_E_BLOCK_CONFIG when the NVRAM block has not immediate priority.⌋*()*

[SWS_NvM_00757] ⌈If development error detection is enabled for NvM module, the function NvM_EraseNvBlock shall report the DET error NVM_E_BLOCK_LOCKED when the block is locked.⌋*()*

[SWS_NvM_00637] ⌈If development error detection is enabled for NvM module, the function NvM_CancelWriteAll shall report the DET error NVM_E_UNINIT when NVM is not yet initialized.⌋*()*

[SWS_NvM_00638] ⌈If development error detection is enabled for NvM module, the function NvM_InvalidateNvBlock shall report the DET error NVM_E_UNINIT when NVM is not yet initialized.⌋*()*

[SWS_NvM_00639] ⌈If development error detection is enabled for NvM module, the function NvM_InvalidateNvBlock shall report the DET error NVM_E_BLOCK_PENDING when NVRAM block identifier is already queued or currently in progress.⌋*()*

[SWS_NvM_00642] ⌈If development error detection is enabled for NvM module, the function NvM_InvalidateNvBlock shall report the DET error NVM_E_PARAM_BLOCK_ID when the passed BlockID is out of range.⌋*()*

[SWS_NvM_00756] ⌈If development error detection is enabled for NvM module, the function NvM_InvalidateNvBlock shall report the DET error NVM_E_BLOCK_LOCKED when the block is locked.⌋*()*

[SWS_NvM_00643] ⌈If development error detection is enabled for NvM module, the function NvM_SetRamBlockStatus shall report the DET error NVM_E_UNINIT when NVM not yet initialized.⌋*()*

[SWS_NvM_00644] ⌈If development error detection is enabled for NvM module, the function NvM_SetRamBlockStatus shall report the DET error NVM_E_BLOCK_PENDING when NVRAM block identifier is already queued or currently in progress.⌋*()*

[SWS_NvM_00645] ⌈If development error detection is enabled for NvM module, the function NvM_SetRamBlockStatus shall report the DET error NVM_E_PARAM_BLOCK_ID when the passed BlockID is out of range.⌋*()*

[SWS_NvM_00758] ⌈If development error detection is enabled for NvM module, the function NvM_SetRamBlockStatus shall report the DET error NVM_E_BLOCK_LOCKED when the block is locked.⌋*()*

[SWS_NvM_00646] ⌈If development error detection is enabled for NvM module, the function NvM_ReadAll shall report the DET error NVM_E_UNINIT when NVM is not yet initialized.⌋*()*

[SWS_NvM_00647] ⌈If development error detection is enabled for NvM module, the function NvM_WriteAll shall report the DET error NVM_E_UNINIT when NVM is not yet initialized.⌋*()*

[SWS_NvM_00648] ⌈If development error detection is enabled for NvM module, the function NvM_CancelJobs shall report the DET error NVM_E_UNINIT when NVM is not yet initialized.⌋*()*

[SWS_NvM_00649] ⌈If development error detection is enabled for NvM module, the function NvM_CancelJobs shall report the DET error NVM_E_PARAM_BLOCK_ID when the passed BlockID is out of range.⌋*()*

[SWS_NvM_00728] ⌈If development error detection is enabled for NvM module, the function NvM_SetBlockLockStatus shall report the DET error NVM_E_UNINIT when NVM is not yet initialized.⌋*()*

[SWS_NvM_00729] ⌈If development error detection is enabled for NvM module, the function NvM_SetBlockLockStatus shall report the DET error NVM_E_BLOCK_PENDING when NVRAM block identifier is already queued or currently in progress.⌋*()*

[SWS_NvM_00730] ⌈If development error detection is enabled for NvM module, the function NvM_SetBlockLockStatus shall report the DET error NVM_E_BLOCK_CONFIG when the NVRAM block is configured with NvMWriteBlockOnce = TRUE.⌋*()*

[SWS_NvM_00731] ⌈If development error detection is enabled for NvM module, the function NvM_SetBlockLockStatus shall report the DET error NVM_E_PARAM_BLOCK_ID when the passed BlockID is out of range.⌋*()*

[SWS_NvM_00863] ⌈If development error detection is enabled for NvM module, the function NvM_ValidateAll shall report the DET error NVM_E_UNINIT when NVM is not yet initialized.⌋*(SRS_Mem_00137)*

[SWS_NvM_00954] ⌈If development error detection is enabled for NvM module, the function NvM_WriteBlock shall report the DET error NVM_E_WRITE_ONCE_STATUS_UNKNOWN when a write request is made for a block configured with NVM_WRITE_BLOCK_ONCE (TRUE) for which no read request was made prior to this.⌋*()*

[SWS_NvM_00955] ⌈If development error detection is enabled for NvM module, the function NvM_WritePRAMBlock shall report the DET error NVM_E_WRITE_ONCE_STATUS_UNKNOWN when a write request is made for a block configured with NVM_WRITE_BLOCK_ONCE (TRUE) for which no read request was made prior to this.⌋*()*

[SWS_NvM_00956] ⌈If development error detection is enabled for NvM module, the job of the function NvM_WriteAll shall report the DET error NVM_E_WRITE_ONCE_STATUS_UNKNOWN when the processing of a block configured with NVM_WRITE_BLOCK_ONCE (TRUE) for which no read request was made prior to this.⌋*()*

[SWS_NvM_00957] ⌈If development error detection is enabled for NvM module, the job of the function NvM_EraseNvBlock shall report the DET error NVM_E_WRITE_ONCE_STATUS_UNKNOWN when a write request is made for a block configured with NVM_WRITE_BLOCK_ONCE (TRUE) for which no read request was made prior to this.⌋*()*

[SWS_NvM_00958] ⌈If development error detection is enabled for NvM module, the job of the function NvM_InvalidateNvBlock shall report the DET error NVM_E_WRITE_ONCE_STATUS_UNKNOWN when a write request is made for a block configured with NVM_WRITE_BLOCK_ONCE (TRUE) for which no read request was made prior to this.⌋*()*

[SWS_NvM_00962] ⌈If development error detection is enabled for NvM module, the function NvM_WriteBlock shall report the DET error NVM_E_WRITE_PROTECTED when the block is write protected.⌋*()*

[SWS_NvM_00963] ⌈If development error detection is enabled for NvM module, the function NvM_WritePRAMBlock shall report the DET error NVM_E_WRITE_PROTECTED when the block is write protected.⌋*()*

[SWS_NvM_00964] ⌈If development error detection is enabled for NvM module, the function NvM_EraseNvBlock shall report the DET error NVM_E_WRITE_PROTECTED when the block is write protected.⌋*()*

[SWS_NvM_00965] ⌈If development error detection is enabled for NvM module, the function NvM_InvalidateNvBlock shall report the DET error NVM_E_WRITE_PROTECTED when the block is write protected.⌋*()*

7.3.2 Runtime Errors

[SWS_NvM_00947] ⌈

Type of error	*Related error code*	*Error value*
NvM queue is full so the request cannot be queued, be the request either standard or immediate.	NVM_E_QUEUE_FULL	0xA0

⌋()

[SWS_NvM_00948] ⌈The run-time error NVM_E_QUEUE_FULL shall be reported to Det, by the NvM module, each time a request cannot be queued because the related queue is full.⌋*(SRS_Mem_00038)*

7.3.3 Transient Faults

There are no transient faults.

7.3.4 Production Errors

7.3.4.1 NVM_E_HARDWARE

[SWS_NvM_00835] ⌈

Error Name:	NVM_E_HARDWARE	
Short Description:	Reading from or writing to non volatile memory failed	
Long Description:	If read job (multi job or single job read) fails either because the MemIf reports MEMIF_JOB_FAILED, MEMIF_BLOCK_INCONSISTENT or a CRC mismatch occurs or if a write/invalidate/erase job fails because the MemIf reports MEMIF_JOB_FAILED, NvM shall report NVM_E_HARDWARE to the DEM.	
Detection Criteria:	Fail	MemIf reports MEMIF_JOB_FAILED, MEMIF_BLOCK_INCONSISTENT or a CRC mismatch occurs during read / write / invalidate / erase operation.
	Pass	Read / write / invalidate / erase is successfull. (MemIf does not report MEMIF_JOB_FAILED , MEMIF_BLOCK_INCONSISTENT and no CRC mismatch occurs)
Secondary Parameters:	The condition under which the FAIL and/or PASS detection is active: Every time a read / write / invalidate / erase is requested for the block NvM shall report if the condition of the block changed.	
Time Required:	Not applicabale. (there is no timeout monitoring in the NvM)	
Monitor Frequency	continous	

⌋()

7.3.5 Extended Production Errors

Type or error	Related error code	Value [hex]
The processing of the read service detects an inconsistency	NVM_E_INTEGRITY_FAILED	Assigned by DEM
The processing of the service fails	NVM_E_REQ_FAILED	Assigned by DEM
The Static Block ID check during read failed	NVM_E_WRONG_BLOCK_ID	Assigned by DEM
The write verification failed	NVM_E_VERIFY_FAILED	Assigned by DEM
There is a loss of redundancy for a block of redundant type	NVM_E_LOSS_OF_REDUNDANCY	Assigned by DEM

[SWS_NvM_00591] ⌈The extended production error NVM_E_INTEGRITY_FAILED (value assigned by DEM, see container NvmDemEventParameterRefs) shall be detectable by the NvM module when API request integrity failed, depending on whether the build version mode is in production mode.⌋*()*

[SWS_NvM_00592] ⌈The extended production error NVM_E_REQ_FAILED (value assigned by DEM, see container NvmDemEventParameterRefs) shall be detectable by the NvM module when API request failed, depending on whether the build version mode is in production mode.⌋*()*

[SWS_NvM_00593] ⌈The extended production error NVM_E_WRONG_BLOCK_ID (value assigned by DEM, see container NvmDemEventParameterRefs) shall be detectable by the NvM module when Static Block ID check failed, depending on whether the build version mode is in production mode.⌋*(SRS_Mem_08555)*

[SWS_NvM_00594] ⌈The extended production error NVM_E_VERIFY_FAILED (value assigned by DEM, see container NvmDemEventParameterRefs) shall be detectable by the NvM module when write Verification failed, depending on whether the build version mode is in production mode.⌋*()*

[SWS_NvM_00595] ⌈The extended production error NVM_E_LOSS_OF_REDUNDANCY (value assigned by DEM, see container NvmDemEventParameterRefs) shall be detectable by the NvM module when loss of redundancy, depending on whether the build version mode is in production mode.⌋*()*

[SWS_NvM_00871] ⌈Each time a request is made to the NvM, the job of that request, if encountering an error situation, shall report the corresponding production error.⌋*()*

7.3.5.1 NVM_E_INTEGRITY_FAILED

Error Name:	NVM_E_INTEGRITY_FAILED

▽

△

Short Description:	Processin of the read service detects an inconsistency.	
Long Description:	If the read for a block detects that the data and/or CRC are corrupted based on the CRC check performed after the read was finished successfully (JobEnd Notification from underlyinh memory module). This only applies for blocks configured with CRC.	
Detection Criteria:	Fail	See SWS_NvM_00864
	Pass	See SWS_NvM_00872
Secondary Parameters:	The condition under which the FAIL or PASS detection is active: CRC checking is performed each time a block with CRC is read successfully by the underlying memory module and it will indicate failure or pass.	
Time Required:	Not applicable. There is no timeout monitoring or constraint for NvM.	
Monitor Frequency	continous	

[SWS_NvM_00864] ⌈Fail condition: NVM_E_INTEGRITY_FAILED is reported by the NvM module if the processing of a read request will detect, via the CRC checking, corruption of the data and/or CRC of the block that was subject to the read operation.⌋ *()*

[SWS_NvM_00872] ⌈Pass condition: when requirement SWS_NvM_00864 does not apply, meaning the data of the block is not corrupted in terms of CRC checking.⌋*()*

7.3.5.2 NVM_E_REQ_FAILED

Error Name:	NVM_E_REQ_FAILED	
Short Description:	Processin of the read service failed at a lower layer in the MemStack architecture, including all retries.	
Long Description:	If the underlying layer reports JobErrorNotification, indicating that the request failed, either after it was accepted by the underlying memory module or because the module refused the request. This is done after all retries also failed.	
Detection Criteria:	Fail	See SWS_NvM_00865
	Pass	See: SWS_NvM_00873
Secondary Parameters:	The condition under which the FAIL or PASS detection is active: check is performed to see if the job was accepted or not and, if accepted, to see if it finished successfully or not.	
Time Required:	Not applicable. There is no timeout monitoring or constraint for NvM.	
Monitor Frequency	continous	

[SWS_NvM_00865] ⌈Fail condition: NVM_E_REQ_FAILED is reported by the NvM module if a user request is either rejected and the number of configured retries expired or if it was accepted and then failed, while being processed by the underlying memory stack module.⌋*()*

[SWS_NvM_00873] ⌈Pass condition: when requirement SWS_NvM_00865 does not apply, meaning that the user request was accepted by the undelying layer, either from the first attempt or from one of the retries, and that it finished successfully, from the point of view of the underlying layer (request result is MEMIF_JOB_OK).⌋*()*

7.3.5.3 NVM_E_WRONG_BLOCK_ID

Error Name:	NVM_E_WRONG_BLOCK_ID	
Short Description:	Static block ID check, during read, indicates failure.	
Long Description:	If the read was successfully finished by the underlying memory module but the Static ID check failed (meaning the block ID that was read is not the same as the block ID for which the read was requested).	
Detection Criteria:	Fail	See SWS_NvM_00866
	Pass	See SWS_NvM_00874
Secondary Parameters:	The condition under which the FAIL or PASS detection is active: check is performed each time the reading of a block is finished successfully by the underlying memory module, if the block is configured to have the Static ID checking performed for it.	
Time Required:	Not applicable. There is no timeout monitoring or constraint for NvM.	
Monitor Frequency	continous	

[SWS_NvM_00866] ⌈Fail condition: NVM_E_WRONG_BLOCK_ID is reported by the NvM module if, after the block data is successfully read from the non-volatile memory, the Static ID that was retrieved is not the same as the current one, for the block the read was requested for.⌋*()*

[SWS_NvM_00874] ⌈Pass condition: when requirement SWS_NvM_00866 does not apply, meaning that the block ID that was read from the non-volatile memory is the same as the block ID for which the read was requested.⌋*()*

7.3.5.4 NVM_E_VERIFY_FAILED

Error Name:	NVM_E_VERIFY_FAILED	
Short Description:	The write verification failed.	
Long Description:	If, after a successfully finished write, the verification for the written data fails.	
Detection Criteria:	Fail	See SWS_NvM_00867
	Pass	See SWS_NvM_00875
Secondary Parameters:	The condition under which the FAIL or PASS detection is active: a check is performed each time a block that is configured to have write verification performed on it, has a write operation successfully finished.	
Time Required:	Not applicable. There is no timeout monitoring or constraint for NvM.	
Monitor Frequency	continous	

[SWS_NvM_00867] ⌈Fail condition: NVM_E_VERIFY_FAILED is reported by the Nv M module if, after a successful write, the write verification indicates failure and the configured number of retries has expired.⌋*()*

[SWS_NvM_00875] ⌈Pass condition: when requirement SWS_NvM_00867 does not apply, meaning that the write verification indicates success, the latest for the last retry attempt.⌋*()*

7.3.5.5 NVM_E_LOSS_OF_REDUNDANCY

Error Name:	NVM_E_LOSS_OF_REDUNDANCY	
Short Description:	A redundant block has lost the redundancy.	
Long Description:	A redundant block has the same contents written in two different block instances - hence the redundancy. If the contents are different, if the first instance becomes corrupted or if the first instance cannot be read then NvM will report this fault.	
Detection Criteria:	Fail	See SWS_NvM_00868
	Pass	See SWS_NvM_00876
Secondary Parameters:	The condition under which the FAIL or PASS detection is active: checks are performed whenever a reading is requested for a redundant block.	
Time Required:	Not applicable. There is no timeout monitoring or constraint for NvM.	
Monitor Frequency	continous	

[SWS_NvM_00868] ⌈Fail condition: NVM_E_LOSS_OF_REDUNDANCY is reported by the NvM module if the reading performed over a REDUNDANT block indicates the block has lost its redundancy.⌋*()*

Note: The loss of redundancy is detected if the reading of the first instance of the block fails and the reading of the second instance of the block is finished successfully.

[SWS_NvM_00876] ⌈Pass condition: when requirement SWS_NvM_00868 does not apply, meaning that the NvM did not detect the loss of redundancy for a REDUNDANT block.⌋*()*

8 API specification

8.1 Imported types

- In this chapter all types included from the following modules are listed:

[SWS_NvM_00446] ⌈

Module	*Header File*	*Imported Type*
Dem	Rte_Dem_Type.h	Dem_EventIdType
	Rte_Dem_Type.h	Dem_EventStatusType
MemIf	MemIf.h	MemIf_JobResultType (obsolete)
	MemIf.h	MemIf_StatusType
Std	Std_Types.h	Std_ReturnType
	Std_Types.h	Std_VersionInfoType

⌋*()*

8.2 Type definitions

8.2.1 NvM_ConfigType

[SWS_NvM_00880] ⌈

Name	NvM_ConfigType	
Kind	Structure	
Elements	implementation specific	
	Type	–
	Comment	–
Description	Configuration data structure of the NvM module.	
Available via	NvM.h	

⌋() Since this type is used for compliance purposes only (meaning that NvM_Init will now have a pointer to this type as parameter, based on SWS_BSW_00047) it will be left to the developer to chose how to implement it, considering it has no use for the Nv M module in any way.

8.2.2 NvM_MultiBlockRequestType

[SWS_NvM_91003] ⌈

Name	NvM_MultiBlockRequestType		
Kind	Enumeration		
Range	NVM_READ_ALL	0x00	NvM_ReadAll was performed
	NVM_WRITE_ALL	0x01	NvM_WriteAll was performed
	NVM_VALIDATE_ALL	0x02	NvM_ValidateAll was performed
	NVM_FIRST_INIT_ALL	0x03	NvM_FirstInitAll was performed
	NVM_CANCEL_WRITE_ ALL	0x04	NvM_CancelWriteAll was performed
Description	Identifies the type of request performed on multi block when signaled via the callback function or when reporting to BswM		
Available via	NvM.h		

⌋()

8.3 Function definitions

8.3.1 Synchronous requests

8.3.1.1 NvM_Init

[SWS_NvM_00447] ⌈

Service Name	NvM_Init	
Syntax	`void NvM_Init ( const NvM_ConfigType* ConfigPtr )`	
Service ID [hex]	0x00	
Sync/Async	Synchronous	
Reentrancy	Non Reentrant	
Parameters (in)	ConfigPtr	Pointer to the selected configuration set.
Parameters (inout)	None	
Parameters (out)	None	
Return value	None	
Description	Service for resetting all internal variables.	
Available via	NvM.h	

⌋(*SRS_BSW_00414, SWS_BSW_00047*)

[SWS_NvM_00881] ⌈The Configuration pointer ConfigPtr shall always have a NULL_PTR value.⌋*()*

The Configuration pointer ConfigPtr is currently not used and shall therefore be set to a NULL_PTR value when calling the NvM_Init API.

[SWS_NvM_00399] ⌈The function NvM_Init shall reset all internal variables, e.g. the queues, request flags, state machines, to their initial values. It shall signal "INIT DONE" internally, e.g. to enable job processing and queue management.⌋*(SRS_BSW_00101, SRS_BSW_00406)*

[SWS_NvM_00400] ⌈The function NvM_Init shall not modify the permanent RAM block contents or call explicit synchronization callback, as this shall be done on NvM_ReadAll.⌋*(SRS_BSW_00101, SRS_BSW_00406)*

[SWS_NvM_00192] ⌈The function NvM_Init shall set the dataset index of all NVRAM blocks of type NVM_BLOCK_DATASET to zero.⌋*()*

[SWS_NvM_00193] ⌈The function NvM_Init shall not initialize other modules (it is assumed that the underlying layers are already initialized).⌋*()*

The function NvM_Init is affected by the common 10.2.2 and published configuration parameter.

Hint: The time consuming NVRAM block initialization and setup according to the block descriptor 10.2.3 shall be done by the NvM_ReadAll request.

8.3.1.2 NvM_SetDataIndex

[SWS_NvM_00448] ⌈

Service Name	NvM_SetDataIndex	
Syntax	`Std_ReturnType NvM_SetDataIndex ( NvM_BlockIdType BlockId, uint8 DataIndex )`	
Service ID [hex]	0x01	
Sync/Async	Synchronous	
Reentrancy	Reentrant	
Parameters (in)	BlockId	The block identifier uniquely identifies one NVRAM block descriptor. A NVRAM block descriptor contains all needed information about a single NVRAM block.
	DataIndex	Index position (association) of a NV/ROM block.
Parameters (inout)	None	
Parameters (out)	None	
Return value	Std_ReturnType	E_OK: The index position was set successfully. E_NOT_OK: An error occurred.
Description	Service for setting the DataIndex of a dataset NVRAM block.	
Available via	NvM.h	

⌋(*SRS_Mem_08007*)

[SWS_NvM_00014] ⌈The function NvM_SetDataIndex shall set the index to access a certain dataset of a NVRAM block (with/without ROM blocks).⌋*()*

[SWS_NvM_00263] ⌈The function NvM_SetDataIndex shall leave the content of the corresponding RAM block unmodified.⌋*()*

[SWS_NvM_00264] ⌈For blocks with block management different from NVM_BLOCK_DATASET, NvM_SetDataIndex shall return without any effect in production mode. Further, E_NOT_OK shall be returned.⌋*()*

[SWS_NvM_00707] ⌈The NvM module's environment shall have initialized the NvM module before it calls the function NvM_SetDataIndex.⌋*()*

8.3.1.3 NvM_GetDataIndex

[SWS_NvM_00449] ⌈

Service Name	NvM_GetDataIndex	
Syntax	`Std_ReturnType NvM_GetDataIndex ( NvM_BlockIdType BlockId, uint8* DataIndexPtr )`	
Service ID [hex]	0x02	
Sync/Async	Synchronous	
Reentrancy	Reentrant	

▽

△

Parameters (in)	BlockId	The block identifier uniquely identifies one NVRAM block descriptor. A NVRAM block descriptor contains all needed information about a single NVRAM block.
Parameters (inout)	None	
Parameters (out)	DataIndexPtr	Pointer to where to store the current dataset index (0..255)
Return value	Std_ReturnType	E_OK: The index position has been retrieved successfully. E_NOT_OK: An error occurred.
Description	Service for getting the currently set DataIndex of a dataset NVRAM block	
Available via	NvM.h	

⌋()

[SWS_NvM_00021] ⌈The function NvM_GetDataIndex shall get the current index (association) of a dataset NVRAM block (with/without ROM blocks).⌋()

[SWS_NvM_00265] ⌈For blocks with block management different from NVM_BLOCK_DATASET, NvM_GetDataIndex shall set the index pointed by DataIndexPtr to zero. Further, E_NOT_OK shall be returned.⌋()

[SWS_NvM_00708] ⌈The NvM module's environment shall have initialized the NvM module before it calls the function NvM_GetDataIndex.⌋()

8.3.1.4 NvM_SetBlockProtection

[SWS_NvM_00450] ⌈

Service Name	NvM_SetBlockProtection	
Syntax	`Std_ReturnType NvM_SetBlockProtection ( NvM_BlockIdType BlockId, boolean ProtectionEnabled )`	
Service ID [hex]	0x03	
Sync/Async	Synchronous	
Reentrancy	Reentrant	
Parameters (in)	BlockId	The block identifier uniquely identifies one NVRAM block descriptor. A NVRAM block descriptor contains all needed information about a single NVRAM block.
	ProtectionEnabled	TRUE: Write protection shall be enabled FALSE: Write protection shall be disabled
Parameters (inout)	None	
Parameters (out)	None	
Return value	Std_ReturnType	E_OK: The block was enabled/disabled as requested E_NOT_OK: An error occured.
Description	Service for setting/resetting the write protection for a NV block.	
Available via	NvM.h	

⌋(*SRS_Mem_00127*)

[SWS_NvM_00016] ⌈The function NvM_SetBlockProtection shall set/reset the write protection for the corresponding NV block by setting the write protection attribute in the administrative part of the corresponding NVRAM block.⌋*(SRS_Mem_00127)*

[SWS_NvM_00709] ⌈The NvM module's environment shall have initialized the NvM module before it calls the function NvM_SetBlockProtection.⌋*()*

8.3.1.5 NvM_GetErrorStatus

[SWS_NvM_00451] ⌈

Service Name	NvM_GetErrorStatus	
Syntax	`Std_ReturnType NvM_GetErrorStatus ( NvM_BlockIdType BlockId, NvM_RequestResultType* RequestResultPtr )`	
Service ID [hex]	0x04	
Sync/Async	Synchronous	
Reentrancy	Reentrant	
Parameters (in)	BlockId	The block identifier uniquely identifies one NVRAM block descriptor. A NVRAM block descriptor contains all needed information about a single NVRAM block.
Parameters (inout)	None	
Parameters (out)	RequestResultPtr	Pointer to where to store the request result. See NvM_Request ResultType .
Return value	Std_ReturnType	E_OK: The block dependent error/status information was read successfully. E_NOT_OK: An error occured.
Description	Service to read the block dependent error/status information.	
Available via	NvM.h	

⌋*(SRS_Mem_00020)*

[SWS_NvM_00015] ⌈The function NvM_GetErrorStatus shall read the block dependent error/status information in the administrative part of a NVRAM block. The status/error information of a NVRAM block shall be set by a former or current asynchronous request.⌋*(SRS_Mem_00020)*

[SWS_NvM_00710] ⌈The NvM module's environment shall have initialized the NvM module before it calls the function NvM_GetErrorStatus.⌋*()*

8.3.1.6 NvM_GetVersionInfo

[SWS_NvM_00452] ⌈

Service Name	NvM_GetVersionInfo	
Syntax	`void NvM_GetVersionInfo ( Std_VersionInfoType* versioninfo )`	
Service ID [hex]	0x0f	
Sync/Async	Synchronous	
Reentrancy	Reentrant	
Parameters (in)	None	
Parameters (inout)	None	
Parameters (out)	versioninfo	Pointer to where to store the version information of this module.
Return value	None	
Description	Service to get the version information of the NvM module.	
Available via	NvM.h	

⌋()

8.3.1.7 NvM_SetRamBlockStatus

[SWS_NvM_00453] ⌈

Service Name	NvM_SetRamBlockStatus	
Syntax	`Std_ReturnType NvM_SetRamBlockStatus ( NvM_BlockIdType BlockId, boolean BlockChanged )`	
Service ID [hex]	0x05	
Sync/Async	Synchronous	
Reentrancy	Reentrant	
Parameters (in)	BlockId	The block identifier uniquely identifies one NVRAM block descriptor. A NVRAM block descriptor contains all needed information about a single NVRAM block.
	BlockChanged	TRUE: Validate the permanent RAM block or the explicit synchronization and mark block as changed. FALSE: Invalidate the permanent RAM block or the explicit synchronization and mark block as unchanged.
Parameters (inout)	None	
Parameters (out)	None	
Return value	Std_ReturnType	E_OK: The status of the permanent RAM block or the explicit synchronization was changed as requested. E_NOT_OK: An error occurred.
Description	Service for setting the RAM block status of a permanent RAM block or the status of the explicit synchronization of a NVRAM block.	
Available via	NvM.h	

⌋(*SRS_Mem_08545*)

[SWS_NvM_00240] ⌈The function NvM_SetRamBlockStatus shall only work on NVRAM blocks with a permanently configured RAM block or on NVRAM blocks config-

ured to support explicit synchronization, that have NvMBlockUseSetRamBlockStatus enabled and shall have no effect to other NVRAM blocks.⌋*(SRS_Mem_08546)*

[SWS_NvM_00241] ⌈The function NvM_SetRamBlockStatus shall assume that a changed permanent RAM block or the content of the RAM mirror in the NvM module (in case of explicit synchronization) is valid (basic assumption).⌋*(SRS_Mem_08545)*

[SWS_NvM_00405] ⌈When the "BlockChanged" parameter passed to the function NvM_SetRamBlockStatus is FALSE the corresponding RAM block is either invalid or unchanged (or both).⌋*(SRS_Mem_08545)*

[SWS_NvM_00406] ⌈When the "BlockChanged" parameter passed to the function NvM_SetRamBlockStatus is TRUE, the corresponding permanent RAM block or the content of the RAM mirror in the NvM module (in case of explicit synchronization) is valid and changed.⌋*()*

[SWS_NvM_00121] ⌈For blocks with a permanently configured RAM, the function NvM_SetRamBlockStatus shall request the recalculation of CRC in the background, i.e. the CRC recalculation shall be processed by the NvM_MainFunction, if the given "BlockChanged" parameter is TRUE and CRC calculation in RAM is configured (i.e. NvMCalcRamBlockCrc == TRUE).⌋*()*

Note: If a block processed by the job of the function NvM_SetRamBlockStatus has explicit synchronization configured for it then the block owner should provide the related RAM data for the comparison. The call made by NvM to the explicit synchronization 'write' callback should be successful.

Hint: In some cases, a permanent RAM block cannot be validated neither by a reload of its NV data, nor by a load of its ROM data during the execution of a NvM_ReadAll command (startup). The application is responsible to fill in proper data to the RAM block and to validate the block via the function NvM_SetRamBlockStatus before this RAM block can be written to its corresponding NV block by NvM_WriteAll.

It is expected that the function NvM_SetRamBlockStatus will be called frequently for NVRAM blocks which are configured to be protected in RAM via CRC. Otherwise this function only needs to be called once to mark a block as "changed" and to be processed during NvM_WriteAll.

[SWS_NvM_00906] ⌈If the function NvM_SetRamBlockStatus is called for a block that does not have permanent RAM but it does have explicit synchronization and the "BlockChanged" parameter is TRUE then the job of the function NvM_SetRamBlockStatus shall use the explicit synchronization callback for data storage (write) in order to obtain the data over which to calculate the CRC for the block.⌋*(SRS_Mem_08550, SRS_Mem_08545, SRS_Mem_00136)*

[SWS_NvM_00907] ⌈If the explicit synchronization callback that is called by the job of the function NvM_SetRamBlockStatus returns E_NOT_OK then NvM shall retry to call the callback for the number of retries that are configured for the explicit synchronization.⌋*(SRS_Mem_08554)*

[SWS_NvM_00908] ⌈If the explicit synchronization callback that is called by the job of the function NvM_SetRamBlockStatus returns E_NOT_OK then NvM shall perform the configured retries, one per NvM_MainFunction call.⌋*(SRS_Mem_08554)*

[SWS_NvM_00909] ⌈If the explicit synchronization callback that is called by the job of the function NvM_SetRamBlockStatus returns E_NOT_OK for the initial call and for all retry attempts then NvM will consider the job completed, keep the block marked as "BlockChanged" and continue as though it finished successfully.⌋*(SRS_Mem_08550, SRS_Mem_08545, SRS_Mem_00136)*

[SWS_NvM_00910] ⌈The function NvM_SetRamBlockStatus shall not change the request result for the block ID received as parameter.⌋*(SRS_Mem_00038)*

[SWS_NvM_00911] ⌈A queued background CRC calculation done by the function NvM_SetRamBlockStatus shall not change the request result for the received block ID.⌋*(SRS_Mem_00038)*

[SWS_NvM_00711] ⌈The NvM module's environment shall have initialized the NvM module before it calls the function NvM_SetRamBlockStatus.⌋*()*

[SWS_NvM_00408] ⌈The NvM module shall provide the function NvM_SetRamBlockStatus only if it is configured via NvMSetRamBlockStatusApi (see 10.2.2).⌋*()*

8.3.1.8 NvM_SetBlockLockStatus

[SWS_NvM_00548] ⌈

Service Name	NvM_SetBlockLockStatus	
Syntax	`void NvM_SetBlockLockStatus ( NvM_BlockIdType BlockId, boolean BlockLocked )`	
Service ID [hex]	0x13	
Sync/Async	Synchronous	
Reentrancy	Reentrant	
Parameters (in)	BlockId	The block identifier uniquely identifies one NVRAM block descriptor. A NVRAM block descriptor contains all needed information about a single NVRAM block.
	BlockLocked	TRUE: Mark the RAM.block as locked FALSE: Mark the RAM.block as unlocked
Parameters (inout)	None	
Parameters (out)	None	
Return value	None	
Description	Service for setting the lock status of a permanent RAM block or of the explicit synchronization of a NVRAM block.	
Available via	NvM.h	

⌋*(SRS_Mem_08546)*

[SWS_NvM_00732] ⌈The function NvM_SetBlockLockStatus shall only work on NVRAM blocks with a permanently configured RAM block or on NVRAM blocks configured to support explicit synchronization and shall have no effect to other NVRAM blocks.
Hint: This function is to be used mainly by DCM, but it can also be used by complex device drivers. The function is not included in the ServicePort interface.⌋*()*

8.3.1.9 NvM_CancelJobs

[SWS_NvM_00535] ⌈

Service Name	NvM_CancelJobs	
Syntax	`Std_ReturnType NvM_CancelJobs ( NvM_BlockIdType BlockId )`	
Service ID [hex]	0x10	
Sync/Async	Synchronous	
Reentrancy	Reentrant	
Parameters (in)	BlockId	The block identifier uniquely identifies one NVRAM block descriptor. A NVRAM block descriptor contains all needed information about a single NVRAM block.
Parameters (inout)	None	
Parameters (out)	None	
Return value	Std_ReturnType	E_OK: The job was successfully removed from queue. E_NOT_OK: The job could not be found in the queue.
Description	Service to cancel all jobs pending for a NV block.	
Available via	NvM.h	

⌋*(SRS_Mem_08560)*

[SWS_NvM_00536] ⌈The function NvM_CancelJobs shall cancel all jobs pending in the queue for the specified NV Block. If requested the result type for the canceled blocks is NVM_REQ_CANCELED.⌋*(SRS_Mem_08560)*

[SWS_NvM_00537] ⌈A currently processed job is not canceled, and shall continue even after the call of NvM_CancelJobs.⌋*()*

[SWS_NvM_00225] ⌈The function NvM_CancelJobs shall set the block specific request result for the specified NVRAM block to NVM_REQ_CANCELED if the request is accepted.⌋*()*

Hint: The intent is just to empty the queue during the cleanup phase in case of termination or restart of a partition, to avoid later end of job notification.

[SWS_NvM_00984] ⌈The function NvM_CancelJobs shall set the canceled block's RAM block state to INVALID/UNCHANGED.⌋*()*

8.3.2 Asynchronous single block requests

8.3.2.1 NvM_ReadBlock

[SWS_NvM_00454] ⌈

Service Name	NvM_ReadBlock	
Syntax	`Std_ReturnType NvM_ReadBlock ( NvM_BlockIdType BlockId, void* NvM_DstPtr )`	
Service ID [hex]	0x06	
Sync/Async	Asynchronous	
Reentrancy	Reentrant	
Parameters (in)	BlockId	The block identifier uniquely identifies one NVRAM block descriptor. A NVRAM block descriptor contains all needed information about a single NVRAM block.
Parameters (inout)	None	
Parameters (out)	NvM_DstPtr	Pointer to the RAM data block.
Return value	Std_ReturnType	E_OK: request has been accepted E_NOT_OK: request has not been accepted
Description	Service to copy the data of the NV block to its corresponding RAM block.	
Available via	NvM.h	

⌋*(SRS_LIBS_08533, SRS_Mem_00016)*

[SWS_NvM_00010] ⌈The job of the function NvM_ReadBlock shall copy the data of the NV block to the corresponding RAM block.⌋*(SRS_Mem_00016)*

Note: The error/status NVM_REQ_OK will be set if the job succeeds. (See SWS_NvM_00270)

[SWS_NvM_00195] ⌈The function NvM_ReadBlock shall take over the given parameters, queue the read request in the job queue and return.⌋*(SRS_Mem_00016)*

[SWS_NvM_00196] ⌈If the function NvM_ReadBlock is provided with a valid RAM block address then it is used.⌋*(SRS_Mem_00016)*

[SWS_NvM_00898] ⌈If the function NvM_ReadBlock is provided with NULL_PTR as a RAM block address and it has a permanent RAM block configured then the permanent RAM block is used.⌋*(SRS_Mem_00016)*

[SWS_NvM_00899] ⌈If the function NvM_ReadBlock is provided with NULL_PTR as a RAM block address and it has the explicit synchronization configured then the explicit synchronization is used.⌋*(SRS_Mem_00016)*

[SWS_NvM_00278] ⌈The job of the function NvM_ReadBlock shall provide the possibility to copy NV data to a temporary RAM block although the NVRAM block is configured with a permanent RAM block or explicit synchronization callbacks. In this case, the parameter NvM_DstPtr has to be unequal to the NULL pointer. Otherwise a DET-Parameter error (see Section 7.3) shall be emitted.⌋*()*

[SWS_NvM_00198] ⌈The function NvM_ReadBlock shall set the RAM block state to INVALID/UNCHANGED immediately when the block is successfully enqueued.⌋*()*

[SWS_NvM_00199] ⌈The job of the function NvM_ReadBlock shall initiate a read attempt on the second NV block if the passed BlockId references a NVRAM block of type NVM_BLOCK_REDUNDANT and the read attempts on the first NV block fail.⌋*()*

[SWS_NvM_00340] ⌈In case of NVRAM block management type NVM_BLOCK_DATASET, the job of the function NvM_ReadBlock shall copy only that NV block to the corresponding RAM block which is selected via the data index in the administrative block.⌋*()*

[SWS_NvM_00355] ⌈The job of the function NvM_ReadBlock shall not copy the NV block to the corresponding RAM block if the NVRAM block management type is NVM_BLOCK_DATASET and the NV block selected by the dataset index is invalidate.⌋*()*

[SWS_NvM_00651] ⌈The job of the function NvM_ReadBlock shall not copy the NV block to the corresponding RAM block if the NVRAM block management type is NVM_BLOCK_DATASET and the NV block selected by the dataset index is inconsistent.⌋*()*

[SWS_NvM_00354] ⌈The job of the function NvM_ReadBlock shall copy the ROM block to RAM and set the request result to NVM_REQ_OK if the NVRAM block management type is NVM_BLOCK_DATASET and the dataset index points at a ROM block.⌋*()*

[SWS_NvM_00200] ⌈The job of the function NvM_ReadBlock shall set the RAM block to valid and assume it to be unchanged after a successful copy process of the NV block to RAM.⌋*()*

[SWS_NvM_00366] ⌈The job of the function NvM_ReadBlock shall set the RAM block to valid and assume it to be changed if the default values are copied to the RAM successfully.⌋*()*

[SWS_NvM_00206] ⌈The job of the function NvM_ReadBlock shall set the request result to NVM_REQ_OK if the NV block was copied successfully from NV memory to RAM.⌋*()*

[SWS_NvM_00341] ⌈The job of the function NvM_ReadBlock shall set the request result to NVM_REQ_NV_INVALIDATED if the MemIf reports MEMIF_BLOCK_INVALID.⌋*()*

[SWS_NvM_00652] ⌈The job of the function NvM_ReadBlock shall report no error to the DEM if the MemIf reports MEMIF_BLOCK_INVALID.⌋*()*

[SWS_NvM_00358] ⌈The job of the function NvM_ReadBlock shall set the request result to NVM_REQ_INTEGRITY_FAILED if:

- the MemIf reports MEMIF_BLOCK_INCONSISTENT and
- NvMRomBlockDataAddress is not configured (no ROM block with default data is available) for the block and

- NvMInitBlockCallback is not configured (no init callback) for the block.

⌋()

[SWS_NvM_00653] ⌈The job of the function NvM_ReadBlock shall report NVM_E_INTEGRITY_FAILED to the DEM if the MemIf reports MEMIF_BLOCK_INCONSISTENT.⌋()

Note: After the production of an ECU / a car, on the production line all blocks shall have been written with valid data (may be default data) and all diagnostic events (errors) shall have been deleted. If the process does not allow to write all NV blocks during production than the NvM will report diagnostic events (errors) because of blocks that were never written and reported as MEMIF_BLOCK_INCONSISTENT by MemIf.

[SWS_NvM_00359] ⌈The job of the function NvM_ReadBlock shall set the request result to NVM_REQ_NOT_OK if:

- the MemIf reports MEMIF_JOB_FAILED and
- NvMRomBlockDataAddress is not configured (no ROM block with default data is available) for the block and
- NvMInitBlockCallback is not configured (no init callback) for the block.

⌋()

[SWS_NvM_00654] ⌈The job of the function NvM_ReadBlock shall report NVM_E_REQ_FAILED to the DEM if the MemIf reports MEMIF_JOB_FAILED.⌋()

[SWS_NvM_00279] ⌈The job of the function NvM_ReadBlock shall set the request result to NVM_REQ_OK if the block management type of the given NVRAM block is NVM_BLOCK_REDUNDANT and one of the NV blocks was copied successfully from NV memory to RAM.⌋()

[SWS_NvM_00655] ⌈The job of the function NvM_ReadBlock shall report no error to the DEM if the block management type of the given NVRAM block is NVM_BLOCK_REDUNDANT and one of the NV blocks was copied successfully from NV memory to RAM.⌋()

[SWS_NvM_00316] ⌈The job of the function NvM_ReadBlock shall mark every NVRAM block that has been configured with NVM_WRITE_BLOCK_ONCE (TRUE) as write protected if that block is valid and with consistent data. This write protection cannot be cleared by NvM_SetBlockProtection.⌋()

[SWS_NvM_00317] ⌈The job of the function NvM_ReadBlock shall invalidate a NVRAM block of management type redundant if both NV blocks have been invalidated.⌋()

[SWS_NvM_00201] ⌈The job of the function NvM_ReadBlock shall request a CRC recalculation over the RAM block data after the copy process [SWS_NvM_00180] if the NV block is configured with CRC, i.e. if NvMCalRamBlockCrC == TRUE for the NV block.⌋()

[SWS_NvM_00202] ⌈The job of the function NvM_ReadBlock shall load the default values according to processing of NvM_RestoreBlockDefaults (also set the request result to NVM_REQ_RESTORED_DEFAULTS) if the recalculated CRC is not equal to the CRC stored in NV memory.⌋*()*

[SWS_NvM_00658] ⌈NvM_ReadBlock: If there are no default values available, the RAM blocks shall remain invalid.⌋*()*

[SWS_NvM_00657] ⌈The job of the function NvM_ReadBlock shall load the default values according to processing of NvM_RestoreBlockDefaults (also set the request result to NVM_REQ_RESTORED_DEFAULTS) if the read request passed to the underlying layer fails (MemIf reports MEMIF_JOB_FAILED or MEMIF_BLOCK_INCONSISTENT) and if the default values are available.⌋*()*

[SWS_NvM_00203] ⌈The job of the function NvM_ReadBlock shall report NVM_E_INTEGRITY_FAILED to the DEM if a CRC mismatch occurs.⌋*()*

[SWS_NvM_00204] ⌈The job of the function NvM_ReadBlock shall set the request result to NVM_REQ_INTEGRITY_FAILED if:

- a CRC mismatch occurs and
- NvMRomBlockDataAddress is not configured (no ROM block with default data is available) for the block and
- NvMInitBlockCallback is not configured (no init callback) for the block.

⌋*()*

[SWS_NvM_00712] ⌈The NvM module's environment shall have initialized the NvM module before it calls the function NvM_ReadBlock.⌋*()*

8.3.2.2 NvM_WriteBlock

[SWS_NvM_00455] ⌈

Service Name	NvM_WriteBlock	
Syntax	`Std_ReturnType NvM_WriteBlock ( NvM_BlockIdType BlockId, const void* NvM_SrcPtr )`	
Service ID [hex]	0x07	
Sync/Async	Asynchronous	
Reentrancy	Reentrant	
Parameters (in)	BlockId	The block identifier uniquely identifies one NVRAM block descriptor. A NVRAM block descriptor contains all needed information about a single NVRAM block.
	NvM_SrcPtr	Pointer to the RAM data block.
Parameters (inout)	None	

▽

△

Parameters (out)	None	
Return value	Std_ReturnType	E_OK: request has been accepted E_NOT_OK: request has not been accepted
Description	Service to copy the data of the RAM block to its corresponding NV block.	
Available via	NvM.h	

⌋*(SRS_Mem_00017)*

[SWS_NvM_00410] ⌈The job of the function NvM_WriteBlock shall copy the data of the RAM block to its corresponding NV block.⌋*(SRS_Mem_00017)*

Note: The error/status NVM_REQ_OK will be set if the job succeeds. (See SWS_NvM_00270)

[SWS_NvM_00749] ⌈The function NvM_WriteBlock shall return with E_NOT_OK, if a locked NVRAM block is referenced by the passed BlockId parameter. and a DET error (see Section 7.3) shall be emitted.⌋*()*

[SWS_NvM_00208] ⌈The function NvM_WriteBlock shall take over the given parameters, queue the write request in the job queue and return.⌋*(SRS_Mem_08541)*

[SWS_NvM_00209] ⌈The function NvM_WriteBlock shall check the NVRAM block protection when the request is enqueued but not again before the request is executed.⌋*()*

[SWS_NvM_00300] ⌈The function NvM_WriteBlock shall cancel a pending job immediately in a destructive way if the passed BlockId references a NVRAM block configured to have immediate priority. The immediate job shall be the next active job to be processed.⌋*()*

[SWS_NvM_00210] ⌈If the function NvM_WriteBlock is provided with a valid RAM block address then it is used.⌋*(SRS_Mem_00017)*

[SWS_NvM_00900] ⌈If the function NvM_WriteBlock is provided with NULL_PTR as a RAM block address and it has a permanent RAM block configured then the permanent RAM block is used.⌋*(SRS_Mem_00017)*

[SWS_NvM_00901] ⌈If the function NvM_WriteBlock is provided with NULL_PTR as a RAM block address and it has the explicit synchronization configured then the explicit synchronization is used.⌋*(SRS_Mem_00017)*

[SWS_NvM_00280] ⌈The job of the function NvM_WriteBlock shall provide the possibility to copy a temporary RAM block to a NV block although the NVRAM block is configured with a permanent RAM block or explicit synchronization callbacks. In this case, the parameter NvM_SrcPtr has to be unequal to a NULL pointer. Otherwise a DET-Parameter error (see Section 7.3) shall be emitted⌋*()*

[SWS_NvM_00212] ⌈The job of the function NvM_WriteBlock shall request a CRC recalculation before the RAM block will be copied to NV memory if the NV block is configured with CRC [SWS_NvM_00180].⌋*()*

[SWS_NvM_00852] ⌈The job of the function NvM_WriteBlock shall skip writing and consider the job as successfully finished if the NvMBlockUseCRCCompMechanism attribute of the NVRAM Block is set to true and the RAM block CRC calculated by the write job is equal to the CRC calculated during the last successful read or write job. This mechanism shall not be applied to blocks for which a loss of redundancy has been detected.⌋*(SRS_Mem_00136)*

[SWS_NvM_00338] ⌈The job of the function NvM_WriteBlock shall copy the RAM block to the corresponding NV block which is selected via the data index in the administrative block if the NVRAM block management type of the given NVRAM block is NVM_BLOCK_DATASET.⌋*()*

[SWS_NvM_00303] ⌈If the block is successfully enqueued and if the RAM block state is VALID/UNCHANGED or INVALID/UNCHANGED, the function NvM_WriteBlock shall set the RAM block state to VALID/CHANGED.⌋*()*

Note: If the block has an explicit synchronization callback (NvM_WriteRamBlockTo Nvm) configured the validation will be done just after NvM_WriteRamBlockToNvm is successfully processed.

Note/Hint: The job of the function NvM_WriteBlock shall assume a referenced permanent RAM block or the RAM mirror in the NvM module in case of explicit synchronization to be valid when the request is passed to the NvM module.

[SWS_NvM_00213] ⌈The job of the function NvM_WriteBlock shall check the number of write retries using a write retry counter to avoid infinite loops. Each negative result reported by the memory interface shall be followed by an increment of the retry counter. In case of a retry counter overrun, the job of the function NvM_WriteBlock shall set the request result to NVM_REQ_NOT_OK.⌋*(SRS_Mem_08554)*

[SWS_NvM_00659] ⌈The job of the function NvM_WriteBlock shall check the number of write retries using a write retry counter to avoid infinite loops. Each negative result reported by the memory interface shall be followed by an increment of the retry counter. In case of a retry counter overrun, the job of the function NvM_WriteBlock shall report NVM_E_REQ_FAILED to the DEM.⌋*()*

[SWS_NvM_00216] ⌈The configuration parameter NVM_MAX_NUM_OF_WRITE_RETRIES shall prescribe the maximum number of write retries for the job of the function NvM_WriteBlock when RAM block data cannot be written successfully to the corresponding NV block (see 10.2.2).⌋*()*

[SWS_NvM_00760] ⌈The job of the function NvM_WriteBlock shall copy the data content of the RAM block to both corresponding NV blocks if the NVRAM block management type of the processed NVRAM block is NVM_BLOCK_REDUNDANT.⌋*()*

[SWS_NvM_00761] ⌈If the processed NVRAM block is of type NVM_BLOCK_REDUNDANT the job of the function NvM_WriteBlock shall start to copy the data of the RAM block to NV block which has not been read during the jobs started by NvM_ReadBlock, NvM_ReadPRAMBlock or NvM_ReadAll then continue to copy the other NV block.⌋*()*

[SWS_NvM_00284] ⌈The job of the function NvM_WriteBlock shall set NVM_REQ_OK as request result if the passed BlockId references a NVRAM block of type NVM_BLOCK_REDUNDANT and at least one of the NV blocks has been written successfully.⌋*()*

[SWS_NvM_00328] ⌈The job of the function NvM_WriteBlock shall set the write protection flag in the administrative block immediately if the NVRAM block is configured with NvMWriteBlockOnce == TRUE and the data has been written successfully to the NV block.⌋*()*

[SWS_NvM_00713] ⌈The NvM module's environment shall have initialized the NvM module before it calls the function NvM_WriteBlock.⌋*()*

Hint: To avoid the situation that in case of redundant NVRAM blocks two different NV blocks are containing different but valid data at the same time, each client of the function NvM_WriteBlock may call NvM_InvalidateNvBlock in advance.

[SWS_NvM_00547] ⌈The job of the function NvM_WriteBlock with Block ID 1 shall write the compiled NVRAM configuration ID to the stored NVRAM configuration ID (block 1).⌋*()*

Hint: If a pristine ECU is flashed for the first time, such a call invoked by will ensure that after a power-off without a proper shutdown, everything is as expected at the next start-up. Otherwise, the new configuration ID would not be stored in NV RAM and all ROM defaultd would be used. A macro scan be used to indicate this usage.

8.3.2.3 NvM_RestoreBlockDefaults

[SWS_NvM_00456] ⌈

Service Name	NvM_RestoreBlockDefaults	
Syntax	`Std_ReturnType NvM_RestoreBlockDefaults ( NvM_BlockIdType BlockId, void* NvM_DestPtr )`	
Service ID [hex]	0x08	
Sync/Async	Asynchronous	
Reentrancy	Non Reentrant	
Parameters (in)	BlockId	The block identifier uniquely identifies one NVRAM block descriptor. A NVRAM block descriptor contains all needed information about a single NVRAM block.
Parameters (inout)	None	
Parameters (out)	NvM_DestPtr	Pointer to the RAM data block.
Return value	Std_ReturnType	E_OK: request has been accepted E_NOT_OK: request has not been accepted
Description	Service to restore the default data to its corresponding RAM block.	
Available via	NvM.h	

⌋*(SRS_Mem_00018)*

[SWS_NvM_00012] ⌈The job of the function NvM_RestoreBlockDefaults shall restore the default data to its corresponding RAM block.⌋*(SRS_Mem_00018)*

Note: The error/status NVM_REQ_OK will be set if the job succeeds. (See SWS_NvM_00270)

[SWS_NvM_00224] ⌈The function NvM_RestoreBlockDefaults shall take over the given parameters, queue the request in the job queue and return.⌋*()*

[SWS_NvM_00267] ⌈The job of the function NvM_RestoreBlockDefaults shall load the default data from a ROM block if a ROM block is configured.⌋*(SRS_Mem_00018)*

[SWS_NvM_00266] ⌈The NvM module's environment shall call the function NvM_RestoreBlockDefaults to obtain the default data if no ROM block is configured for a NVRAM block and an application callback routine is configured via the parameter NvMInitBlockCallback.⌋*(SRS_Mem_00018)*

[SWS_NvM_00353] ⌈The function NvM_RestoreBlockDefaults shall return with E_NOT_OK if the block management type of the given NVRAM block is NVM_BLOCK_DATASET, at least one ROM block is configured and the data index points at a NV block.⌋*()*

[SWS_NvM_00435] ⌈If the function NvM_RestoreBlockDefaults is provided with a valid RAM block address then it is used.⌋*(SRS_Mem_00018)*

[SWS_NvM_00902] ⌈If the function NvM_RestoreBlockDefaults is provided with NULL_PTR as a RAM block address and it has a permanent RAM block configured then the permanent RAM block is used.⌋*(SRS_Mem_00018)*

[SWS_NvM_00903] ⌈If the function NvM_RestoreBlockDefaults is provided with NULL_PTR as a RAM block address and it has the explicit synchronization configured then the explicit synchronization is used.⌋*(SRS_Mem_00018)*

[SWS_NvM_00436] ⌈The NvM module's environment shall pass a pointer unequal to NULL via the parameter NvM_DstPtr to the function NvM_RestoreBlockDefaults in order to copy ROM data to a temporary RAM block although the NVRAM block is configured with a permanent RAM block or explicit synchronization callbacks. Otherwise a DET-Parameter error (see Section 7.3) shall be emitted⌋*()*

[SWS_NvM_00227] ⌈The function NvM_RestoreBlockDefaults shall set the RAM block state to INVALID/UNCHANGED immediately when the block is successfully enqueued.⌋*()*

[SWS_NvM_00228] ⌈The job of the function NvM_RestoreBlockDefaults shall validate and assume a RAM block to be changed if the requested RAM block is permanent or after explicit synchronization callback (NvMReadRamBlockFromNvCallback) that is called returns E_OK and the copy process of the default data to RAM was successful .⌋*()*

[SWS_NvM_00229] ⌈The job of the function NvM_RestoreBlockDefaults shall request a recalculation of CRC from a RAM block after the copy process/validation if a CRC is configured for this RAM block.⌋*()*

[SWS_NvM_00714] ⌈The NvM module's environment shall have initialized the NvM module before it calls the function NvM_RestoreBlockDefaults.⌋*()*

Hint: For the block management type NVM_BLOCK_DATASET, the application has to ensure that a valid dataset index is selected (pointing to ROM data).

[SWS_NvM_00883] ⌈If the block has no ROM default data and no NvMInitBlockCallback configured for it then the function NvM_RestoreBlockDefaults shall leave the block status unchanged and return E_NOT_OK as result.⌋*()*

[SWS_NvM_00885] ⌈If the block has no default data, it has no InitBlockCallbackFunction configured and the development error detection is enabled then the NvM_Restore BlockDefaults API shall report the error NVM_E_BLOCK_WITHOUT_DEFAULTS error to the Det module.⌋*()*

8.3.2.4 NvM_EraseNvBlock

[SWS_NvM_00457] ⌈

Service Name	NvM_EraseNvBlock	
Syntax	`Std_ReturnType NvM_EraseNvBlock ( NvM_BlockIdType BlockId )`	
Service ID [hex]	0x09	
Sync/Async	Asynchronous	
Reentrancy	Reentrant	
Parameters (in)	BlockId	The block identifier uniquely identifies one NVRAM block descriptor. A NVRAM block descriptor contains all needed information about a single NVRAM block.
Parameters (inout)	None	
Parameters (out)	None	
Return value	Std_ReturnType	E_OK: request has been accepted E_NOT_OK: request has not been accepted
Description	Service to erase a NV block.	
Available via	NvM.h	

⌋*(SRS_Mem_08544)*

[SWS_NvM_00415] ⌈The job of the function NvM_EraseNvBlock shall erase a NV block.⌋*(SRS_Mem_08544)*

Note: The error/status NVM_REQ_OK will be set if the job succeeds. (See SWS_NvM_00270)

[SWS_NvM_00231] ⌈The function NvM_EraseNvBlock shall take over the given parameters, queue the request and return.⌋*()*

[SWS_NvM_00418] ⌈The function NvM_EraseNvBlock shall queue the request to erase in case of disabled write protection.⌋*()*

[SWS_NvM_00416] ⌈The job of the function NvM_EraseNvBlock shall leave the content of the RAM block unmodified.⌋*()*

[SWS_NvM_00959] ⌈The job of the function NvM_EraseNvBlock shall leave the write protection unchanged for the blocks configured with NVM_WRITE_BLOCK_ONCE (TRUE).⌋*()*

[SWS_NvM_00661] ⌈The function NvM_EraseNvBlock shall return with E_NOT_OK if a ROM block of a dataset NVRAM block is referenced.⌋*()*

[SWS_NvM_00662] ⌈NvM_EraseNvBlock: The NvM module shall not re-check the write protection before fetching the job from the job queue.⌋*()*

[SWS_NvM_00269] ⌈If the referenced NVRAM block is of type NVM_BLOCK_REDUNDANT, the function NvM_EraseNvBlock shall only succeed when both NV blocks have been erased.⌋*()*

[SWS_NvM_00271] ⌈The job of the function NvM_EraseNvBlock shall set the request result to NVM_REQ_NOT_OK if the processing of the service fails.⌋*()*

[SWS_NvM_00663] ⌈The job of the function NvM_EraseNvBlock shall report NVM_E_REQ_FAILED to the DEM if the processing of the service fails.⌋*()*

[SWS_NvM_00357] ⌈The function NvM_EraseNvBlock shall return with E_NOT_OK, when development error detection is enabled and the referenced NVRAM block is configured with standard priority.⌋*()*

[SWS_NvM_00715] ⌈The NvM module's environment shall have initialized the NvM module before it calls the function NvM_EraseNvBlock.⌋*()*

8.3.2.5 NvM_InvalidateNvBlock

[SWS_NvM_00459] ⌈

Service Name	NvM_InvalidateNvBlock	
Syntax	`Std_ReturnType NvM_InvalidateNvBlock ( NvM_BlockIdType BlockId )`	
Service ID [hex]	0x0b	
Sync/Async	Asynchronous	
Reentrancy	Reentrant	
Parameters (in)	BlockId	The block identifier uniquely identifies one NVRAM block descriptor. A NVRAM block descriptor contains all needed information about a single NVRAM block.
Parameters (inout)	None	
Parameters (out)	None	
Return value	Std_ReturnType	E_OK: request has been accepted E_NOT_OK: request has not been accepted

▽

△

Description	Service to invalidate a NV block.
Available via	NvM.h

⌋*(SRS_Mem_08011)*

[SWS_NvM_00421] ⌈The job of the function NvM_InvalidateNvBlock shall invalidate a NV block.⌋*(SRS_Mem_08011)*

Note: The error/status NVM_REQ_OK will be set if the job succeeds. (See SWS_NvM_00270)

[SWS_NvM_00422] ⌈The job of the function NvM_InvalidateNvBlock shall leave the RAM block unmodified.⌋*()*

[SWS_NvM_00960] ⌈The job of the function NvM_InvalidateNvBlock shall leave the write protection unchanged for the blocks configured with NVM_WRITE_BLOCK_ONCE (TRUE).⌋*()*

[SWS_NvM_00424] ⌈The function NvM_InvalidateNvBlock shall queue the request if the write protection of the corresponding NV block is disabled.⌋*()*

[SWS_NvM_00239] ⌈The function NvM_InvalidateNvBlock shall take over the given parameters, queue the request and return.⌋*()*

[SWS_NvM_00664] ⌈The function NvM_InvalidateNvBlock shall return with E_NOT_OK if a ROM block of a dataset NVRAM block is referenced by the BlockId parameter.⌋*()*

[SWS_NvM_00665] ⌈The NvM module shall not recheck write protection before fetching the job from the job queue.⌋*()*

[SWS_NvM_00274] ⌈If the referenced NVRAM block is of type NVM_BLOCK_REDUNDANT, the function NvM_InvalidateNvBlock shall only set the request result NvM_RequestResultType to NVM_REQ_OK when both NV blocks have been invalidated.⌋*()*

[SWS_NvM_00275] ⌈The function NvM_InvalidateNvBlock shall set the request result to NVM_REQ_NOT_OK if the processing of this service fails.⌋*()*

[SWS_NvM_00666] ⌈The function NvM_InvalidateNvBlock shall report NVM_E_REQ_FAILED to the DEM if the processing of this service fails.⌋*()*

[SWS_NvM_00717] ⌈The NvM module's environment shall have initialized the NvM module before it calls the function function NvM_InvalidateNvBlock.⌋*()*

8.3.2.6 NvM_ReadPRAMBlock

[SWS_NvM_00764] ⌈

Service Name	NvM_ReadPRAMBlock	
Syntax	`Std_ReturnType NvM_ReadPRAMBlock ( NvM_BlockIdType BlockId )`	
Service ID [hex]	0x16	
Sync/Async	Asynchronous	
Reentrancy	Reentrant	
Parameters (in)	BlockId	The block identifier uniquely identifies one NVRAM block descriptor. A NVRAM block descriptor contains all needed information about a single NVRAM block.
Parameters (inout)	None	
Parameters (out)	None	
Return value	Std_ReturnType	E_OK: request has been accepted E_NOT_OK: request has not been accepted
Description	Service to copy the data of the NV block to its corresponding permanent RAM block.	
Available via	NvM.h	

⌋*(SRS_LIBS_08533, SRS_Mem_00016)*

[SWS_NvM_00765] ⌈The job of the function NvM_ReadPRAMBlock shall copy the data of the NV block to the permanent RAM block.⌋*(SRS_Mem_00016)*

Note: The error/status NVM_REQ_OK will be set if the job succeeds. (See SWS_NvM_00270)

[SWS_NvM_00766] ⌈The function NvM_ReadPRAMBlock shall take over the given parameters, queue the read request in the job queue and return.⌋*(SRS_Mem_00016)*

[SWS_NvM_00767] ⌈If the block is successfully enqueued, the function NvM_ReadPRAMBlock shall set the RAM block state to INVALID/UNCHANGED.⌋*()*

[SWS_NvM_00768] ⌈The job of the function NvM_ReadPRAMBlock shall initiate a read attempt on the second NV block if the passed BlockId references a NVRAM block of type NVM_BLOCK_REDUNDANT and the read attempts on the first NV block fail.⌋*()*

[SWS_NvM_00769] ⌈In case of NVRAM block management type NVM_BLOCK_DATASET, the job of the function NvM_ReadPRAMBlock shall copy only that NV block to the corresponding RAM block which is selected via the data index in the administrative block.⌋*()*

[SWS_NvM_00770] ⌈The job of the function NvM_ReadPRAMBlock shall not copy the NV block to the corresponding RAM block if the NVRAM block management type is NVM_BLOCK_DATASET and the NV block selected by the dataset index is invalidate.⌋*()*

[SWS_NvM_00771] ⌈The job of the function NvM_ReadPRAMBlock shall not copy the NV block to the corresponding RAM block if the NVRAM block management type is NVM_BLOCK_DATASET and the NV block selected by the dataset index is inconsistent.⌋*()*

[SWS_NvM_00772] ⌈The job of the function NvM_ReadPRAMBlock shall copy the ROM block to RAM and set the request result to NVM_REQ_OK if the NVRAM block management type is NVM_BLOCK_DATASET and the dataset index points at a ROM block.⌋*()*

[SWS_NvM_00773] ⌈The job of the function NvM_ReadPRAMBlock shall set the RAM block to valid and assume it to be unchanged after a successful copy process of the NV block to RAM.⌋*()*

[SWS_NvM_00774] ⌈The job of the function NvM_ReadPRAMBlock shall set the RAM block to valid and assume it to be changed if the default values are copied to the RAM successfully.⌋*()*

[SWS_NvM_00775] ⌈The job of the function NvM_ReadPRAMBlock shall set the request result to NVM_REQ_OK if the NV block was copied successfully from NV memory to RAM.⌋*()*

[SWS_NvM_00776] ⌈The job of the function NvM_ReadPRAMBlock shall set the request result to NVM_REQ_NV_INVALIDATED if the MemIf reports MEMIF_BLOCK_INVALID.⌋*()*

[SWS_NvM_00777] ⌈The job of the function NvM_ReadPRAMBlock shall report no error to the DEM if the MemIf reports MEMIF_BLOCK_INVALID.⌋*()*

[SWS_NvM_00778] ⌈The job of the function NvM_ReadPRAMBlock shall set the request result to NVM_REQ_INTEGRITY_FAILED if the MemIf reports MEMIF_BLOCK_INCONSISTENT.⌋*()*

[SWS_NvM_00779] ⌈The job of the function NvM_ReadPRAMBlock shall report NVM_E_INTEGRITY_FAILED to the DEM if the MemIf reports MEMIF_BLOCK_INCONSISTENT.⌋*()*

[SWS_NvM_00780] ⌈The job of the function NvM_ReadPRAMBlock shall set the request result to NVM_REQ_NOT_OK if the MemIf reports MEMIF_JOB_FAILED.⌋*()*

[SWS_NvM_00781] ⌈The job of the function NvM_ReadPRAMBlock shall report NVM_E_REQ_FAILED to the DEM if the MemIf reports MEMIF_JOB_FAILED.⌋*()*

[SWS_NvM_00782] ⌈The job of the function NvM_ReadPRAMBlock shall set the request result to NVM_REQ_OK if the block management type of the given NVRAM block is NVM_BLOCK_REDUNDANT and one of the NV blocks was copied successfully from NV memory to RAM.⌋*()*

[SWS_NvM_00783] ⌈The job of the function NvM_ReadPRAMBlock shall report no error to the DEM if the block management type of the given NVRAM block is NVM_BLOCK_REDUNDANT and one of the NV blocks was copied successfully from NV memory to RAM.⌋*()*

[SWS_NvM_00784] ⌈The job of the function NvM_ReadPRAMBlock shall mark every NVRAM block that has been configured with NVM_WRITE_BLOCK_ONCE (TRUE) as write protected if that block is valid and with consistent data. This write protection cannot be cleared by NvM_SetBlockProtection.⌋*()*

[SWS_NvM_00785] ⌈The job of the function NvM_ReadPRAMBlock shall invalidate a NVRAM block of management type redundant if both NV blocks have been invalidated.⌋*()*

[SWS_NvM_00786] ⌈The job of the function NvM_ReadPRAMBlock shall request a CRC recalculation over the RAM block data after the copy process [SWS_NvM_00180] if the NV block is configured with CRC, i.e. if NvMCalRamBlockCrC == TRUE for the NV block.⌋*()*

[SWS_NvM_00787] ⌈The job of the function NvM_ ReadPRAMBlock shall load the default values according to processing of NvM_RestorePRAMBlockDefaults if the recalculated CRC is not equal to the CRC stored in NV memory.⌋*()*

[SWS_NvM_00788] ⌈NvM_ReadPRAMBlock: If there are no default values available, the RAM blocks shall remain invalid.⌋*()*

[SWS_NvM_00789] ⌈The job of the function NvM_ReadPRAMBlock shall load the default values according to processing of NvM_RestorePRAMBlockDefaults if the read request passed to the underlying layer fails.⌋*()*

[SWS_NvM_00790] ⌈The job of the function NvM_ReadPRAMBlock shall report NVM_E_INTEGRITY_FAILED to the DEM if a CRC mismatch occurs.⌋*()*

[SWS_NvM_00791] ⌈The job of the function NvM_ReadPRAMBlock shall set the request result NVM_REQ_INTEGRITY_FAILED if a CRC mismatch occurs.⌋*()*

[SWS_NvM_00792] ⌈The NvM module's environment shall have initialized the NvM module before it calls the function NvM_ReadPRAMBlock.⌋*()*

[SWS_NvM_00882] ⌈The job of the function NvM_ReadPRAMBlock shall load the default values according to processing of NvM_RestorePRAMBlockDefaults (also set the request result to NVM_REQ_RESTORED_DEFAULTS) if the read request passed to the underlying layer fails (MemIf reports MEMIF_JOB_FAILED or MEMIF_BLOCK_INCONSISTENT) and if the default values are available.⌋*()*

8.3.2.7 NvM_WritePRAMBlock

[SWS_NvM_00793] ⌈

Service Name	NvM_WritePRAMBlock
Syntax	`Std_ReturnType NvM_WritePRAMBlock ( NvM_BlockIdType BlockId )`
Service ID [hex]	0x17
Sync/Async	Asynchronous
Reentrancy	Reentrant

▽

△

Parameters (in)	BlockId	The block identifier uniquely identifies one NVRAM block descriptor. A NVRAM block descriptor contains all needed information about a single NVRAM block.
Parameters (inout)	None	
Parameters (out)	None	
Return value	Std_ReturnType	E_OK: request has been accepted E_NOT_OK: request has not been accepted
Description	Service to copy the data of the permanent RAM block to its corresponding NV block.	
Available via	NvM.h	

⌋*(SRS_Mem_00017)*

[SWS_NvM_00794] ⌈The job of the function NvM_WritePRAMBlock shall copy the data of the permanent RAM block to its corresponding NV block.⌋*(SRS_Mem_00017)*

Note: The error/status NVM_REQ_OK will be set if the job succeeds. (See SWS_NvM_00270)

[SWS_NvM_00797] ⌈The function NvM_WritePRAMBlock shall return with E_NOT_OK, if a locked NVRAM block is referenced by the passed BlockId parameter. and a DET error (see Section 7.3) shall be emitted.⌋*()*

[SWS_NvM_00798] ⌈The function NvM_WritePRAMBlock shall take over the given parameters, queue the write request in the job queue and return.⌋*(SRS_Mem_08541)*

[SWS_NvM_00799] ⌈The function NvM_WritePRAMBlock shall check the NVRAM block protection when the request is enqueued but not again before the request is executed.⌋*()*

[SWS_NvM_00800] ⌈The function NvM_WritePRAMBlock shall cancel a pending job immediately in a destructive way if the passed BlockId references a NVRAM block configured to have immediate priority. The immediate job shall be the next active job to be processed.⌋*()*

[SWS_NvM_00801] ⌈The job of the function NvM_WritePRAMBlock shall request a CRC recalculation before the RAM block will be copied to NV memory if the NV block is configured with CRC [SWS_NvM_00180].⌋*()*

[SWS_NvM_00853] ⌈The job of the function NvM_WritePRAMBlock shall skip writing and consider the job as successfully finished if the NvMBlockUseCRCCompMechanism attribute of the NVRAM Block is set to true and the RAM block CRC calculated by the write job is equal to the CRC calculated during the last successful read or write job. This mechanism shall not be applied to blocks for which a loss of redundancy has been detected.⌋*(SRS_Mem_00136)*

[SWS_NvM_00802] ⌈The job of the function NvM_WritePRAMBlock shall copy the RAM block to the corresponding NV block which is selected via the data index in the administrative block if the NVRAM block management type of the given NVRAM block is NVM_BLOCK_DATASET.⌋*()*

[SWS_NvM_00803] ⌈If the block is successfully enqueued and if the RAM block state is VALID/UNCHANGED or INVALID/UNCHANGED, the function NvM_WritePRAMBlock shall set the RAM block state to VALID/CHANGED.⌋*()*

Note: If the block has an explicit synchronization callback (NvM_WriteRamBlockTo Nvm) configured the validation will be done just after NvM_WriteRamBlockToNvm is successfully processed.

Note/Hint: The job of the function NvM_WritePRAMBlock shall assume a referenced permanent RAM block or the RAM mirror in the NvM module in case of explicit synchronization to be valid when the request is passed to the NvM module.

[SWS_NvM_00804] ⌈The job of the function NvM_WritePRAMBlock shall check the number of write retries using a write retry counter to avoid infinite loops. Each negative result reported by the memory interface shall be followed by an increment of the retry counter. In case of a retry counter overrun, the job of the function NvM_Write PRAMBlock shall set the request result to NVM_REQ_NOT_OK.⌋*(SRS_Mem_08554)*

[SWS_NvM_00805] ⌈The job of the function NvM_WritePRAMBlock shall check the number of write retries using a write retry counter to avoid infinite loops. Each negative result reported by the memory interface shall be followed by an increment of the retry counter. In case of a retry counter overrun, the job of the function NvM_Write PRAMBlock shall report NVM_E_REQ_FAILED to the DEM.⌋*()*

[SWS_NvM_00806] ⌈The configuration parameter NVM_MAX_NUM_OF_WRITE_ RETRIES shall prescribe the maximum number of write retries for the job of the function NvM_WritePRAMBlock when RAM block data cannot be written successfully to the corresponding NV block (see 10.2.2).⌋*()*

[SWS_NvM_00807] ⌈The job of the function NvM_WritePRAMBlock shall copy the data content of the RAM block to both corresponding NV blocks if the NVRAM block management type of the processed NVRAM block is NVM_BLOCK_REDUNDANT.⌋*()*

[SWS_NvM_00808] ⌈If the processed NVRAM block is of type NVM_BLOCK_REDUNDANT the job of the function NvM_WritePRAMBlock shall start to copy the data of the RAM block to NV block which has not been read during the jobs started by NvM_Read Block, NvM_ReadPRAMBlock or NvM_ReadAll then continue to copy the other NV block.⌋*()*

[SWS_NvM_00809] ⌈The job of the function NvM_WritePRAMBlock shall set NVM_ REQ_OK as request result if the passed BlockId references a NVRAM block of type NVM_BLOCK_REDUNDANT and at least one of the NV blocks have been written successfully.⌋*()*

[SWS_NvM_00810] ⌈The job of the function NvM_WritePRAMBlock shall set the write protection flag in the administrative block immediately if the NVRAM block is configured with NvMWriteBlockOnce == TRUE and the data has been written successfully to the NV block.⌋*()*

[SWS_NvM_00811] ⌈The NvM module's environment shall have initialized the NvM module before it calls the function NvM_WritePRAMBlock.⌋*()*

Hint: To avoid the situation that in case of redundant NVRAM blocks two different NV blocks are containing different but valid data at the same time, each client of the function NvM_WritePRAMBlock may call NvM_InvalidateNvBlock in advance.

[SWS_NvM_00812] ⌈The job of the function NvM_WritePRAMBlock with Block ID 1 shall write the compiled NVRAM configuration ID to the stored NVRAM configuration ID (block 1).⌋*()*

Hint: If a pristine ECU is flashed for the first time, such a call invoked by will ensure that after a power-off without a proper shutdown, everything is as expected at the next start-up. Otherwise, the new configuration ID would not be stored in NV RAM and all ROM defaultd would be used. A macro scan be used to indicate this usage.

8.3.2.8 NvM_RestorePRAMBlockDefaults

[SWS_NvM_00813] ⌈

Service Name	NvM_RestorePRAMBlockDefaults	
Syntax	`Std_ReturnType NvM_RestorePRAMBlockDefaults ( NvM_BlockIdType BlockId )`	
Service ID [hex]	0x18	
Sync/Async	Asynchronous	
Reentrancy	Non Reentrant	
Parameters (in)	BlockId	The block identifier uniquely identifies one NVRAM block descriptor. A NVRAM block descriptor contains all needed information about a single NVRAM block.
Parameters (inout)	None	
Parameters (out)	None	
Return value	Std_ReturnType	E_OK: request has been accepted E_NOT_OK: request has not been accepted
Description	Service to restore the default data to its corresponding permanent RAM block.	
Available via	NvM.h	

⌋*(SRS_Mem_00018)*

[SWS_NvM_00814] ⌈The job of the function NvM_RestorePRAMBlockDefaults shall restore the default data to its corresponding permanent RAM block.⌋*(SRS_Mem_00018)*

Note: The error/status NVM_REQ_OK will be set if the job succeeds. (See SWS_NvM_00270)

[SWS_NvM_00815] ⌈The function NvM_RestorePRAMBlockDefaults shall take over the given parameters, queue the request in the job queue and return.⌋*()*

[SWS_NvM_00816] ⌈The job of the function NvM_RestorePRAMBlockDefaults shall load the default data from a ROM block if a ROM block is configured.⌋*(SRS_Mem_00018)*

[SWS_NvM_00817] ⌈The NvM module's environment shall call the function NvM_RestorePRAMBlockDefaults to obtain the default data if no ROM block is configured for a NVRAM block and an application callback routine is configured via the parameter NvMInitBlockCallback.⌋*(SRS_Mem_00018)*

[SWS_NvM_00818] ⌈The function NvM_RestorePRAMBlockDefaults shall return with E_NOT_OK if the block management type of the given NVRAM block is NVM_BLOCK_DATASET, at least one ROM block is configured and the data index points at a NV block.⌋*()*

[SWS_NvM_00819] ⌈The function NvM_RestorePRAMBlockDefaults shall set the RAM block state to INVALID/UNCHANGED immediately when the block is successfully enqueued.⌋*()*

[SWS_NvM_00820] ⌈The job of the function NvM_RestorePRAMBlockDefaults shall validate and assume a RAM block to be changed if the requested RAM block is permanent or after explicit synchronization callback (NvMReadRamBlockFromNvCallback) that is called returns E_OK and the copy process of the default data to RAM was successful .⌋*()*

[SWS_NvM_00821] ⌈The job of the function NvM_RestorePRAMBlockDefaults shall request a recalculation of CRC from a RAM block after the copy process/validation if a CRC is configured for this RAM block.⌋*()*

[SWS_NvM_00822] ⌈The NvM module's environment shall have initialized the NvM module before it calls the function NvM_RestorePRAMBlockDefaults.⌋*()*

Hint: For the block management type NVM_BLOCK_DATASET, the application has to ensure that a valid dataset index is selected (pointing to ROM data).

[SWS_NvM_00884] ⌈If the block has no ROM default data and no NvMInitBlockCallback configured for it then the function NvM_RestorePRAMBlockDefaults shall leave the block status unchanged and return E_NOT_OK as result.⌋*()*

[SWS_NvM_00886] ⌈If the block has no default data, it has no InitBlockCallbackFunction configured and the development error detection is enabled then the NvM_RestorePRAMBlockDefaults API shall report the error NVM_E_BLOCK_WITHOUT_DEFAULTS error to the Det module.⌋*()*

8.3.3 Asynchronous multi block requests

8.3.3.1 NvM_ReadAll

[SWS_NvM_00460] ⌈

Service Name	NvM_ReadAll

▽

△

Syntax	`void NvM_ReadAll ( void )`
Service ID [hex]	0x0c
Sync/Async	Asynchronous
Reentrancy	Non Reentrant
Parameters (in)	None
Parameters (inout)	None
Parameters (out)	None
Return value	None
Description	Initiates a multi block read request.
Available via	NvM.h

⌋*(SRS_LIBS_08533)*

[SWS_NvM_00356] ⌈The multi block service NvM_ReadAll shall provide two distinct functionalities.

- Initialize the management data for all NVRAM blocks (see SWS_NvM_00304 ff)
- Copy data to the permanent RAM blocks or call explicit synchronization callback(NvM_ReadRamBlockFromNvm) for those NVRAM blocks which are configured accordingly.

Note: The two functionalities can be implemented in one loop.⌋*()*

[SWS_NvM_00243] ⌈The function NvM_ReadAll shall signal the request to the NvM module and return. The NVRAM Manager shall defer the processing of the requested ReadAll until all single block job queues are empty.⌋*()*

[SWS_NvM_00304] ⌈The job of the function NvM_ReadAll shall set each proceeding block specific request result for NVRAM blocks in advance.⌋*()*

[SWS_NvM_00667] ⌈The job of the function NvM_ReadAll shall set the multi block request result to NVM_REQ_PENDING in advance.⌋*()*

[SWS_NvM_00895] ⌈The job of the function NvM_ReadAll shall set the multi block request result to NVM_REQ_OK if no NVRAM block processing fails.⌋*(SRS_Mem_00020)*

Note: When the result of an individual block processing (in the context of a multi-block job) is different than NVM_REQ_OK, the individual block processing is considered as failed.

[SWS_NvM_00244] ⌈The job of the function NvM_ReadAll shall iterate over all user NVRAM blocks, i.e. except for reserved Block Ids 0 (multi block request result) and 1 (NV configuration ID), beginning with the lowest Block Id.⌋*()*

[SWS_NvM_00245] ⌈Blocks of management type NVM_BLOCK_DATASET shall not be loaded automatically upon start-up. Thus the selection of blocks, which belong to

block management type NVM_BLOCK_DATASET, shall not be possible for the service NvM_ReadAll.⌋()

[SWS_NvM_00362] ⌈The NvM module shall initiate the recalculation of the RAM CRC for every NVRAM block with a valid permanent RAM block or explicit synchronization callback configured and NvmCalcRamBlockCrc == TRUE during the processing of Nv M_ReadAll.⌋()

Note:

If a block processed by the job of the function NvM_ReadAll has explicit synchronization configured for it then the block owner should provide the related RAM data for the comparison. The call made by NvM to the explicit synchronization 'write' callback should be successful.

[SWS_NvM_00364] ⌈The job of the function NvM_ReadAll shall treat the data for every recalculated RAM CRC which matches the stored RAM CRC as valid and set the block specific request result to NVM_REQ_OK.⌋()

Note: This mechanism enables the NVRAM Manager to avoid overwriting of maybe still valid RAM data with outdated NV data.

[SWS_NvM_00246] ⌈The job of the function NvM_ReadAll shall validate the configuration ID by comparing the stored NVRAM configuration ID vs. the compiled NVRAM configuration ID.⌋()

[SWS_NvM_00669] ⌈NvM_ReadAll: The NVRAM block with the block ID 1 (redundant type with CRC) shall be reserved to contain the stored NVRAM configuration ID.⌋()

[SWS_NvM_00247] ⌈The job of the function NvM_ReadAll shall process the normal runtime preparation for all configured NVRAM blocks in case of configuration ID match.⌋()

[SWS_NvM_00670] ⌈The job of the function NvM_ReadAll shall set the error/status information field of the corresponding NVRAM block's administrative block to NVM_REQ_OK in case of configuration ID match.⌋()

[SWS_NvM_00305] ⌈The job of the function NvM_ReadAll shall report the extended production error NVM_E_REQ_FAILED to the DEM if the configuration ID cannot be read because of an error detected by one of the subsequent SW layers.⌋()

[SWS_NvM_00671] ⌈The job of the function NvM_ReadAll shall set the error status field of the reserved NVRAM block to NVM_REQ_INTEGRITY_FAILED if the configuration ID cannot be read because of an error detected by one of the subsequent SW layers. The NvM module shall behave in the same way as if a configuration ID mismatch was detected.⌋()

[SWS_NvM_00307] ⌈The job of the function NvM_ReadAll shall set the error/status information field of the reserved NVRAM block with ID 1 to NVM_REQ_NOT_OK in the case of configuration ID mismatch.⌋()

[SWS_NvM_00306] ⌈In case the NvM module can not read the configuration ID because the corresponding NV blocks are empty or invalidated, the job of the function NvM_ReadAll shall not report an extended production error or a production error to the DEM.⌋*()*

[SWS_NvM_00672] ⌈In case the NvM module can not read the configuration ID because the corresponding NV blocks are empty or invalidated, the job of the function NvM_ReadAll shall set the error/status information field in this NVRAM block's administrative block to NVM_REQ_NV_INVALIDATED.⌋*()*

[SWS_NvM_00673] ⌈NvM_ReadAll: In case the NvM module can not read the configuration ID because the corresponding NV blocks are empty or invalidated, NVM module shall update the configuration ID from the RAM block assigned to the reserved NVRAM block with ID 1 according to the new (compiled) configuration ID. The NvM module shall behave the same way as if the configuration ID matched.⌋*()*

[SWS_NvM_00248] ⌈The job of the function NvM_ReadAll shall ignore a configuration ID mismatch and behave normal if NvMDynamicConfiguration == FALSE (see 10.2.2).⌋*()*

[SWS_NvM_00249] ⌈The job of the function NvM_ReadAll shall process an extended runtime preparation for all blocks which are configured with NvMResistantToChangedSw == FALSE and NvMDynamicConfiguration == TRUE and configuration ID mismatch occurs.⌋*()*

[SWS_NvM_00674] ⌈The job of the function NvM_ReadAll shall process the normal runtime preparation of all NVRAM blocks when they are configured with NvMResistantToChangedSw == TRUE and NvMDynamicConfiguration == TRUE and if a configuration ID mismatch occurs.⌋*()*

[SWS_NvM_00314] ⌈The job of the function NvM_ReadAll shall mark every NVRAM block that has been configured with NVM_WRITE_BLOCK_ONCE (TRUE) as write protected if that block is valid and with consistent data. This write protection cannot be cleared by NvM_SetBlockProtection.⌋*()*

[SWS_NvM_00315] ⌈The job of the function NvM_ReadAll shall only invalidate a NVRAM block of management type NVM_BLOCK_REDUNDANT if both NV blocks have been invalidated.⌋*()*

[SWS_NvM_00718] ⌈The NvM module's environment shall use the multi block request NvM_ReadAll to load and validate the content of configured permanent RAM or to do the explicit synchronization for configured blocks during start-up [SWS_NvM_00091].⌋*()*

[SWS_NvM_00118] ⌈The job of the function NvM_ReadAll shall process only the permanent RAM blocks or call explicit synchronization callback (NvM_ReadRamBlockFromNvm) for blocks which are configured with NvmSelectBlockForReadall == TRUE.⌋*()*

[SWS_NvM_00287] ⌈The job of the function NvM_ReadAll shall set the request result to NVM_REQ_BLOCK_SKIPPED for each NVRAM block configured to be processed

by the job of the function NvM_ReadAll (NvMSelectBlockForReadAll is checked) and which has not been read during processing of the NvM_ReadAll job.⌋()

[SWS_NvM_00426]{OBSOLETE} ⌈If configured by NvMDrvModeSwitch, the job of the function NvM_ReadAll shall switch the mode of each memory device to "fast-mode" before starting to iterate over all user NVRAM blocks.⌋()

[SWS_NvM_00427]{OBSOLETE} ⌈If configured by NvMDrvModeSwitch, the job of the function NvM_ReadAll shall switch the mode of each memory device to "slow-mode" after having processed all user NVRAM blocks.⌋()

[SWS_NvM_00308] ⌈The job of the function NvM_ReadAll shall restore the default data to the corresponding RAM blocks either if configured by the parameter NvMRom BlockDataAddress or by the parameter NvMInitBlockCallback, and set the error/status field in the administrative block to NVM_REQ_RESTORED_DEFAULTS when processing the extended runtime preparation.⌋()

[SWS_NvM_00309] ⌈When executing the extended runtime preparation, the job of the function NvM_ReadAll shall treat the affected NVRAM blocks as invalid or blank in order to allow rewriting of blocks configured with NVM_BLOCK_WRITE_ONCE == TRUE.⌋()

[SWS_NvM_00310] ⌈The job of the function NvM_ReadAll shall update the configuration ID from the RAM block assigned to the reserved NVRAM block with ID 1 according to the new (compiled) configuration ID, mark the NVRAM block to be written during Nv M_WriteAll and request a CRC recalculation if a configuration ID mismatch occurs and if the NVRAM block is configured with NvMDynamicConfiguration == TRUE.⌋()

[SWS_NvM_00311] ⌈The NvM module shall allow applications to send any request for the reserved NVRAM Block ID 1 if (and only if) NvMDynamicConfiguration is set to TRUE, including NvM_WriteBlock and NvM_WritePRAMBlock.⌋()

[SWS_NvM_00312] ⌈The NvM module shall not send a request for invalidation of the reserved configuration ID NVRAM block to the underlying layer, unless requested so by the application. This shall ensure that the NvM module's environment can rely on this block to be only invalidated at the first start-up of the ECU or if desired by the application.⌋()

[SWS_NvM_00313] ⌈In case of a Configuration ID match, the job of the function Nv M_ReadAll shall not automatically write to the Configuration ID block stored in the reserved NVRAM block 1.⌋()

[SWS_NvM_00288] ⌈The job of the function NvM_ReadAll shall initiate a read attempt on the second NV block for each NVRAM block of type NVM_BLOCK_REDUNDANT [SWS_NvM_00118], where the read attempt of the first block fails (see also SWS_Nv M_00531).⌋()

[SWS_NvM_00290] ⌈The job of the function NvM_ReadAll shall set the block specific request result to NVM_REQ_OK if the job has successfully copied the corresponding NV block from NV memory to RAM.⌋()

[SWS_NvM_00342] ⌈The job of the function NvM_ReadAll shall set the block specific request result to NVM_REQ_NV_INVALIDATED if the MemIf reports MEMIF_BLOCK_INVALID.⌋*()*

[SWS_NvM_00676] ⌈The job of the function NvM_ReadAll shall report no error to the DEM if the MemIf reports MEMIF_BLOCK_INVALID.⌋*()*

[SWS_NvM_00360] ⌈The job of the function NvM_ReadAll shall set the block specific request result to NVM_REQ_INTEGRITY_FAILED if:

- - the MemIf reports MEMIF_BLOCK_INCONSISTENT and
- - NvMRomBlockDataAddress is not configured (no ROM block with default data is available) for the block and
- - NvMInitBlockCallback is not configured (no init callback) for the block.

⌋*()*

[SWS_NvM_00677] ⌈The job of the function NvM_ReadAll shall report NVM_E_INTEGRITY_FAILED to the DEM if the MemIf reports MEMIF_BLOCK_INCONSISTENT.⌋*()*

Note: After the production of an ECU / a car, on the production line all blocks shall have been written with valid data (may be default data) and all diagnostic events (errors) shall have been deleted. If the process does not allow to write all NV blocks during production than the NvM will report diagnostic events (errors) because of blocks that were never written and reported as MEMIF_BLOCK_INCONSISTENT by MemIf.

[SWS_NvM_00361] ⌈The job of the function NvM_ReadAll shall set the block specific request result to NVM_REQ_NOT_OK if:

- - the MemIf reports MEMIF_JOB_FAILED and
- - NvMRomBlockDataAddress is not configured (no ROM block with default data is available) for the block and
- - NvMInitBlockCallback is not configured (no init callback) for the block.

⌋*()*

[SWS_NvM_00678] ⌈The job of the function NvM_ReadAll shall report NVM_E_REQ_FAILED to the DEM, if the MemIf reports MEMIF_JOB_FAILED.⌋*()*

[SWS_NvM_00291] ⌈The job of the function NvM_ReadAll shall set the block specific request result to NVM_REQ_OK if the corresponding block management type is NVM_BLOCK_REDUNDANT and the function has successfully copied one of the NV blocks from NV memory to RAM.⌋*()*

[SWS_NvM_00292] ⌈The job of the function NvM_ReadAll shall request a CRC recalculation over the RAM block data after the copy process SWS_NvM_00180 if the NV block is configured with CRC, , i.e. if NvMCalRamBlockCrC == TRUE for the NV block.⌋*()*

[SWS_NvM_00293] ⌈The job of the function NvM_ReadAll shall load the default values to the RAM blocks according to the processing of NvM_RestoreBlockDefaults (also set the corresponding request result to NVM_REQ_RESTORED_DEFAULTS):

- - if the recalculated CRC is not equal to the CRC stored in NV memory and if the
- default values are available, or
- - if the blocks are marked as invalid (MemIf reports MEMIF_BLOCK_INVALID) and the default values are available.
-

⌋*()*

[SWS_NvM_00679] ⌈The job of the function NvM_ReadAll shall load the default values to the RAM blocks according to the processing of NvM_RestoreBlockDefaults (also set the request result to NVM_REQ_RESTORED_DEFAULTS) if the read request passed to the underlying layer fails (MemIf reports MEMIF_JOB_FAILED or MEMIF_BLOCK_INCONSISTENT) and if the default values are available.⌋*()*

[SWS_NvM_00680] ⌈NvM_ReadAll: If the read request passed to the underlying layer fails and there are no default values available, the job shall leave the RAM blocks invalid.⌋*()*

[SWS_NvM_00294] ⌈The job of the function NvM_ReadAll shall report NVM_E_INTEGRITY_FAILED to the DEM if a CRC mismatch occurs.⌋*()*

[SWS_NvM_00295] ⌈The job of the function NvM_ReadAll shall set a block specific request result to NVM_REQ_INTEGRITY_FAILED if:

- - a CRC mismatch occurs and
- - NvMRomBlockDataAddress is not configured (no ROM block with default data is available) for the block and
- - NvMInitBlockCallback is not configured (no init callback) for the block.

⌋*()*

[SWS_NvM_00302] ⌈The job of the function NvM_ReadAll shall report NVM_E_REQ_FAILED to the DEM if the referenced NVRAM Block is not configured with CRC and the corresponding job process has failed.⌋*()*

[SWS_NvM_00301] ⌈The job of the function NvM_ReadAll shall set the multi block request result to NVM_REQ_NOT_OK if the processing of at least one NVRAM block fails.⌋*()*

Note: When the result of an individual block processing (in the context of a multi-block job) is different than NVM_REQ_OK, the individual block processing (or individual job) is considered as failed.

[SWS_NvM_00281] ⌈If configured by NvMSingleBlockCallback, the job of the function NvM_ReadAll shall call the single block callback after having completely processed

a NVRAM block. For the last block, NvMSingleBlockCallback (if configured) is called before MultiBlockCallback.⌋()

Note: The idea behind using the single block callbacks also for NvM_ReadAll is to speed up the software initialization process:

A single-block callback issued from an NvM_ReadAll will result in an RTE event.

If the RTE is initialized after or during the asynchronous NvM_ReadAll, all or some of these RTE events will get lost because they are overwritten during the RTE initialization (see SWS_Rte_2536).

After its initialization, the RTE can use the "surviving" RTE events to start software components even before the complete NvM_ReadAll has been finished.

For those RTE events that got lost during the initialization: the RTE will start those software components and the software components either query the status of the NV block they want to access or request that NV block to be read. This is exactly the same behavior if the single-block callbacks would not be used in NvM_ReadAll.

[SWS_NvM_00251] ⌈The job of the function NvM_ReadAll shall mark a NVRAM block as "valid/unmodified" if NV data has been successfully loaded to the RAM Block.⌋()

[SWS_NvM_00367] ⌈The job of the function NvM_ReadAll shall set a RAM block to valid and assume it to be changed if the job has successfully copied default values to the corresponding RAM.⌋()

[SWS_NvM_00719] ⌈The NvM module's environment shall have initialized the NvM module before it calls the function NvM_ReadAll.⌋()

The DEM shall already be able to accept error notifications.

[SWS_NvM_00968] ⌈The job of the function NvM_ReadAll shall set the block specific request result to NVM_REQ_INTEGRITY_FAILED if:

- the extended runtime preparation is executed for the block
- NvMRomBlockDataAddress is not configured (no ROM block with default data is available) for the block and
- NvMInitBlockCallback is not configured (no init callback) for the block.
-

⌋()

[SWS_NvM_00970] ⌈The multi block service NvM_ReadAll shall only be provided by the NvM master.⌋()

8.3.3.2 NvM_WriteAll

[SWS_NvM_00461] ⌈

Service Name	NvM_WriteAll
Syntax	`void NvM_WriteAll ( void )`
Service ID [hex]	0x0d
Sync/Async	Asynchronous
Reentrancy	Non Reentrant
Parameters (in)	None
Parameters (inout)	None
Parameters (out)	None
Return value	None
Description	Initiates a multi block write request.
Available via	NvM.h

⌋(*SRS_LIBS_08535*)

[SWS_NvM_00018] ⌈The job of the function NvM_WriteAll shall synchronize the contents of permanent RAM blocks to their corresponding NV blocks or call explicit synchronization callback (NvM_WriteRamBlockToNvm) on shutdown.⌋(*SRS_LIBS_08535*)

[SWS_NvM_00733] ⌈If NVRAM block ID 1 (which holds the configuration ID of the memory layout) is marked as "to be written during NvM_WriteAll", the job of the function NvM_WriteAll shall write this block in a final step (last write operation) to prevent memory layout mismatch in case of a power loss failure during write operation.⌋()

[SWS_NvM_00254] ⌈The function NvM_WriteAll shall signal the request to the NvM module and return. The NVRAM Manager shall defer the processing of the requested WriteAll until all single block job queues are empty.⌋()

[SWS_NvM_00549] ⌈The job of the function NvM_ WriteAll shall set each proceeding block specific request result for NVRAM blocks and the multi block request result to NVM_REQ_PENDING in advance.⌋()

[SWS_NvM_00896] ⌈The job of the function NvM_WriteAll shall set the multi block request result to NVM_REQ_OK if no NVRAM block processing fails.⌋(*SRS_Mem_00020*)

Note: When the result of an individual block processing (in the context of a multi-block job) is different than NVM_REQ_OK, the individual block processing is considered as failed.

[SWS_NvM_00252] ⌈The job of the function NvM_WriteAll shall process only the permanent RAM blocks or call explicit synchronization callback (NvM_WriteRamBlockTo Nvm) for all blocks for which the corresponding NVRAM block parameter NvMSelect BlockForWriteAll is configured to true.⌋()

[SWS_NvM_00430]{OBSOLETE} ⌈If configured by NvMDrvModeSwitch, the job of the function NvM_WriteAll shall set the mode of each memory device to "fast-mode" before starting to iterate over all non-reserved NVRAM blocks.⌋*()*

[SWS_NvM_00431]{OBSOLETE} ⌈If configured by NvMDrvModeSwitch, the job of the function NvM_WriteAll shall set the mode of each memory device to "slow-mode" after having processed all non-reserved NVRAM blocks.⌋*()*

[SWS_NvM_00681]{OBSOLETE} ⌈If configured by NvMDrvModeSwitch, the job of the function NvM_WriteAll shall set the mode of each memory device to "slow-mode" after the function NvM_CancelWriteAll has canceled the job.⌋*()*

[SWS_NvM_00432] ⌈The job of the function NvM_WriteAll shall check the write-protection for each RAM block in advance.⌋*()*

[SWS_NvM_00682] ⌈The job of the function NvM_WriteAll shall check the "valid/modified" state for each RAM block in advance.⌋*()*

[SWS_NvM_00433] ⌈The job of the function NvM_WriteAll shall only write the content of a RAM block to its corresponding NV block for non write-protected NVRAM blocks.⌋*()*

[SWS_NvM_00474] ⌈The job of the function NvM_WriteAll shall correct the redundant data (if configured) if the redundancy has been lost. In this case the job of the function NvM_WriteAll shall ignore write protection for this block in order to be able to repair it.⌋*()*

Note: If NvM implementation detects loss of redundancy during read operation the user (application) should ensure that redundant block is read (e.g. during NvM_ReadAll by configuring the block to be read during NvM_ReadAll). If the block is not read then the NVM will not be able to correct the redundant block's data.

[SWS_NvM_00434] ⌈The job of the function NvM_WriteAll shall skip every write-protected NVRAM block without error notification.⌋*()*

[SWS_NvM_00750] ⌈The job of the function NvM_WriteAll shall skip every locked NVRAM block without error notification.⌋*()*

[SWS_NvM_00854] ⌈The job of the function NvM_WriteAll shall skip an NVRAM block if the NvMBlockUseCRCCompMechanism attribute is set to true and the RAM block CRC calculated by the write job is equal to the CRC calculated during the last successful read or write job. This mechanism shall not be applied to blocks for which a loss of redundancy has been detected.⌋*(SRS_Mem_00136)*

[SWS_NvM_00298] ⌈The job of the function NvM_WriteAll shall set the request result to NVM_REQ_BLOCK_SKIPPED for each NVRAM block configured to be processed by the job of the function NvM_WriteAll (NvMSelectBlockForWriteAll is checked) and which has not been written during processing of the NvM_WriteAll job.⌋*()*

[SWS_NvM_00339] ⌈In case of NVRAM block management type NVM_BLOCK_DATASET, the job of the function NvM_WriteAll shall copy only the RAM block to

the corresponding NV block which is selected via the data index in the administrative block.⌋*()*

[SWS_NvM_00253] ⌈The job of the function NvM_WriteAll shall request a CRC recalculation and renew the CRC from a NVRAM block before writing the data if a CRC is configured for this NVRAM block.⌋*(SRS_LIBS_08535)*

[SWS_NvM_00296] ⌈The job of the function NvM_WriteAll shall check the number of write retries by a write retry counter to avoid infinite loops. Each unsuccessful result reported by the MemIf module shall be followed by an increment of the retry counter.⌋*()*

[SWS_NvM_00683] ⌈The job of the function NvM_WriteAll shall set the block specific request result to NVM_REQ_NOT_OK if the write retry counter becomes greater than the configured NVM_MAX_NUM_OF_WRITE_RETRIES.⌋*()*

[SWS_NvM_00684] ⌈The job of the function NvM_WriteAll shall report NVM_E_REQ_FAILED to the DEM if the write retry counter becomes greater than the configured NVM_MAX_NUM_OF_WRITE_RETRIES.⌋*()*

[SWS_NvM_00762] ⌈The job of the function NvM_WriteAll shall copy the data content of the RAM block to both corresponding NV blocks if the NVRAM block management type of the processed NVRAM block is NVM_BLOCK_REDUNDANT.⌋*()*

[SWS_NvM_00763] ⌈If the processed NVRAM block is of type NVM_BLOCK_REDUNDANT the job of the function NvM_WriteAll shall start to copy the data of the RAM block to NV block which has _not_ been read during the jobs started by NvM_ReadBlock, NvM_ReadPRAMBlock or NvM_ReadAll then continue to copy the other NV block.⌋*()*

[SWS_NvM_00337] ⌈The job of the function NvM_WriteAll shall set the single block request result to NVM_REQ_OK if the processed NVRAM block is of type NVM_BLOCK_REDUNDANT and at least one of the NV blocks has been written successfully.⌋*()*

[SWS_NvM_00238] ⌈The job of the function NvM_WriteAll shall complete the job in a non-destructive way for the NVRAM block currently being processed if a cancellation of NvM_WriteAll is signaled by a call of NvM_CancelWriteAll.⌋*()*

[SWS_NvM_00237] ⌈The NvM module shall set the multi block request result to NVM_REQ_CANCELED in case of cancellation of NvM_WriteAll.⌋*()*

[SWS_NvM_00685] ⌈NvM_WriteAll: The NvM module shall anyway report the error code condition, due to a failed NVRAM block write, to the DEM.⌋*()*

[SWS_NvM_00318] ⌈The job of the function NvM_WriteAll shall set the multi block request result to NVM_REQ_NOT_OK if the processing of at least one NVRAM block fails.⌋*()*

Note: When the result of an individual block processing (in the context of a multi-block job) is different than NVM_REQ_OK, the individual block processing is considered as failed.

[SWS_NvM_00329] ⌈If the job of the function NvM_WriteAll has successfully written data to NV memory for a NVRAM block configured with NvMWriteBlockOnce == TRUE, the job shall immediately set the corresponding write protection flag in the administrative block.⌋*()*

[SWS_NvM_00720] ⌈The NvM module's environment shall have initialized the NvM module before it calls the function NvM_WriteAll.⌋*()*

No other multiblock request shall be pending when the NvM module's environment calls the function NvM_WriteAll.

Note: To avoid the situation that in case of redundant NVRAM blocks two different NV blocks are containing different but valid data at the same time, each client of the NvM_WriteAll service may call NvM_InvalidateNvBlock in advance.

[SWS_NvM_00971] ⌈The multi block service NvM_WriteAll shall only be provided by the NvM master.⌋*()*

8.3.3.3 NvM_CancelWriteAll

[SWS_NvM_00458] ⌈

Service Name	NvM_CancelWriteAll
Syntax	`void NvM_CancelWriteAll ( void )`
Service ID [hex]	0x0a
Sync/Async	Asynchronous
Reentrancy	Non Reentrant
Parameters (in)	None
Parameters (inout)	None
Parameters (out)	None
Return value	None
Description	Service to cancel a running NvM_WriteAll request.
Available via	NvM.h

⌋*(SRS_Mem_08558, SRS_Mem_08540)*

[SWS_NvM_00019] ⌈The function NvM_CancelWriteAll shall cancel a running NvM_WriteAll request. It shall terminate the NvM_WriteAll request in a way that the data consistency during processing of a single NVRAM block is not compromised⌋*(SRS_Mem_08540)*

[SWS_NvM_00232] ⌈The function NvM_CancelWriteAll shall signal the request to the NvM module and return.⌋*()*

[SWS_NvM_00233] ⌈The function NvM_CancelWriteAll shall be without any effect if no NvM_WriteAll request is pending.⌋*()*

[SWS_NvM_00234] ⌈The function NvM_CancelWriteAll shall treat multiple requests to cancel a running NvM_WriteAll request as one request, i.e. subsequent requests will be ignored.⌋*()*

[SWS_NvM_00235] ⌈The request result of the function NvM_CancelWriteAll shall be implicitly given by the result of the NvM_WriteAll request to be canceled.⌋*()*

[SWS_NvM_00236] ⌈The function NvM_CancelWriteAll shall only modify the error/status attribute field of the pending blocks to NVM_REQ_CANCELED and for the currently written block after the processing of a single NVRAM block is finished to NVM_REQ_OK or NVM_REQ_NOT_OK depending on the success of the write operation.⌋*()*

[SWS_NvM_00716] ⌈The NvM module's environment shall have initialized the NvM module before it calls the function function NvM_CancelWriteAll.⌋*()*

[SWS_NvM_00420] ⌈The function NvM_CancelWriteAll shall signal the NvM module and shall not be queued, i.e. there can be only one pending request of this type.⌋*()*

8.3.3.4 NvM_ValidateAll

[SWS_NvM_00855] ⌈

Service Name	NvM_ValidateAll
Syntax	`void NvM_ValidateAll ( void )`
Service ID [hex]	0x19
Sync/Async	Asynchronous
Reentrancy	Non Reentrant
Parameters (in)	None
Parameters (inout)	None
Parameters (out)	None
Return value	None
Description	Initiates a multi block validation request.
Available via	NvM.h

⌋*(SRS_Mem_00137)*

[SWS_NvM_00856] ⌈If auto validation is configured for an NVRAM Block (NvMBlock UseAutoValidation == TRUE), the function NvM_ValidateAll shall set the RAM Block status to "VALID / CHANGED".⌋*(SRS_Mem_00137)*

[SWS_NvM_00857] ⌈For blocks which meet the conditions of SWS_NvM_00856 and in addition have CRC calculation in RAM configured the function NvM_ValidateAll shall request the recalculation of CRC in the background, i.e. the CRC recalculation shall be processed by the NvM_MainFunction.⌋*(SRS_Mem_00137)*

[SWS_NvM_00858] ⌈The function NvM_ValidateAll shall signal the request to the NvM module and return. The NVRAM Manager shall defer the processing of the requested

NvM_ValidateAll function until all single block job queues are empty.⌋(*SRS_Mem_00137*)

[SWS_NvM_00859] ⌈The NvM module's environment shall have initialized the NvM module before it calls the function NvM_ValidateAll.⌋(*SRS_Mem_00137*)

[SWS_NvM_00860] ⌈The job of the function NvM_ValidateAll shall process only the permanent RAM blocks or call explicit synchronization callback (NvM_WriteRamBlock ToNvm) for all blocks for which the corresponding NVRAM Block parameter NvMBlock UseAutoValidation is configured to true.⌋(*SRS_Mem_00137*)

[SWS_NvM_00861] ⌈The job of the function NvM_ValidateAll shall set each proceeding block specific request result for NVRAM blocks and the multi block request result to NVM_REQ_PENDING in advance.⌋(*SRS_Mem_00137*)

[SWS_NvM_00862] ⌈The job of the function NvM_ValidateAll shall set the block specific request result to NVM_REQ_OK if the RAM block was successfully validated.⌋ (*SRS_Mem_00137*)

8.3.3.5 NvM_FirstInitAll

[SWS_NvM_91001] ⌈

Service Name	NvM_FirstInitAll

Syntax	`void NvM_FirstInitAll (`
	`  void`
	`)`

Service ID [hex]	0x14
Sync/Async	Asynchronous
Reentrancy	Non Reentrant
Parameters (in)	None
Parameters (inout)	None
Parameters (out)	None
Return value	None
Description	The function initiates a multi block first initialization request. The job of the function does not care if a block exists in the non-volatile memory or not OR if it is valid (i.e. not corrupted) or not, when processing it.
Available via	NvM.h

⌋()

For each processed block, the job of the function will either write it with default data (if it is not of type DATASET and it has default data) or invalidate the block (if it is of type DATASET or without default data).

The term "default data" means the data from the ROM block (if any) or the one provided inside the InitBlockCallback (if any) by the related block owner.

If NvM_FirstInitAll is called after NvM_ReadAll, then an inconsistency between the Nv M user's expectation of RAM block contents and the actual RAM block contents can occur. Even worse, also concurrent writes to the RAM block content from NvM user side and NvM_FirstInitAll could occur. Hence, calling NvM_FirstInitAll after NvM_Read All should generally be avoided.

In light of the above, the following requirements apply:

[SWS_NvM_00912] ⌈The job of the function NvM_FirstInitAll shall also process the block with ID 1 (which holds the configuration ID of the NvM module), if this block has been configured to be processed by it and dynamic configuration is enabled.⌋*()*

[SWS_NvM_00913] ⌈If a block of type NATIVE that is processed by the NvM_FirstInit All has default data, the NV block shall be written with its default data.⌋*()*

[SWS_NvM_00914] ⌈If a NATIVE block that is processed by the NvM_FirstInitAll has neither permanent RAM nor explicit synchronization then that block shall be processed by using the internal NvM buffer as its RAM and, upon processing, its RAM block state shall be left untouched.⌋*()*

[SWS_NvM_00915] ⌈If a NATIVE block that has either permanent RAM or explicit synchronization is processed by the NvM_FirstInitAll and the block has default data (ROM or Init Callback) then the blocks RAM will be updated with the default data, just like for the processing of a NvM_RestoreBlockDefaults request.⌋*()*

[SWS_NvM_00916] ⌈If a block of type REDUNDANT that is processed by the NvM_ FirstInitAll has default data, both block instances shall be written with that default data.⌋ *()*

[SWS_NvM_00917] ⌈If a REDUNDANT block that is processed by the NvM_FirstInit All has neither permanent RAM nor explicit synchronization then that block shall be processed by using the internal NvM buffer as its RAM and, upon processing, its RAM block state shall be left untouched.⌋*()*

[SWS_NvM_00918] ⌈If a REDUNDANT block that has either permanent RAM or explicit synchronization is processed by the NvM_FirstInitAll and the block has default data (ROM or Init Callback) then the blocks RAM will be updated with the default data, just like for the processing of a NvM_RestoreBlockDefaults request.⌋*()*

[SWS_NvM_00919] ⌈If a block of type NATIVE that is processed by the NvM_FirstInit All does not have default data, the block shall be invalidated using the same mechanism as for NvM_InvalidateNvBlock.⌋*()*

[SWS_NvM_00920] ⌈If a block of type REDUNDANT that is processed by the NvM_ FirstInitAll does not have default data, both block instances shall be invalidated using the same mechanism as for NvM_InvalidateNvBlock.⌋*()*

[SWS_NvM_00921] ⌈If a NATIVE block that is processed by the NvM_FirstInitAll has only the Init Callback configured and the return value of the callback is not E_OK then the job of the function NvM_FirstInitAll shall invalidate the block.⌋*()*

[SWS_NvM_00922] ⌈If a REDUNDANT block that is processed by the NvM_FirstInit All has only the Init Callback configured and the return value of the callback is not E_ OK then the job of the function NvM_FirstInitAll shall invalidate both instances of the block.⌋*()*

Note: An Init Callback returning something else than E_OK is interpreted as a runtime decision of the block owner not to provide default data via this callback. In this case, in order for the state of the block not to remain ambiguous, it is invalidated.

[SWS_NvM_00923] ⌈The job of the function NvM_FirstInitAll shall invalidate all blocks that are of type DATASET and that have been configured to be processed by it.⌋*()*

[SWS_NvM_00924] ⌈The job of the function NvM_FirstInitAll shall invalidate all NV block instances of a block of type DATASET, if the block was configured to be processed by it.⌋*()*

[SWS_NvM_00925] ⌈If the writing of a block of type NATIVE with its default data fails, the job of the function NvM_FirstInitAll shall set the request result to NVM_REQ_NOT_ OK.⌋*()*

[SWS_NvM_00926] ⌈If the writing of a block of type REDUNDANT with its default data fails for both instances, the job of the function NvM_FirstInitAll shall set the request result to NVM_REQ_NOT_OK.⌋*()*

[SWS_NvM_00927] ⌈If the invalidation of a block of type NATIVE fails, the job of the function NvM_FirstInitAll shall set the request result to NVM_REQ_NOT_OK.⌋*()*

[SWS_NvM_00928] ⌈If the invalidation of a block of type REDUNDANT fails for at least one of the two block instances, the job of the function NvM_FirstInitAll shall set the request result to NVM_REQ_NOT_OK.⌋*()*

Note: Since the purpose of the FirstInitAll is to have all selected NvM blocks in a well defined state (either written successfully with the default data or invalidated), if one of the two duplicates of the REDUNDANT block was not invalidated successfully, this has to be known. This is not like the "write" case (see requirements SWS_NvM_00284 and SWS_NvM_00274 for more details).

[SWS_NvM_00929] ⌈If the invalidation of a block of type DATASET fails for at least one of its NV block instances then the job of the function NvM_FirstInitAll shall set the request result to NVM_REQ_NOT_OK.⌋*()*

Note: Since the purpose of the FirstInitAll is to have all selected NvM blocks in a well defined state if at least one of the NV block instances of the DATASET block was not invalidated successfully, this has to be known. The NvM_FirstInitAll processing of blocks of type DATASET implies invalidating all NV block instances of all processed blocks of type DATASET.

[SWS_NvM_00930] ⌈Blocks without permanent RAM block and without explicit synchronization can be configured to be processed by the NvM_FirstInitAll.⌋*()*

[SWS_NvM_00931] ⌈The write protection status of a block shall be completely ignored by the NvM_FirstInitAll functionality.⌋*()*

Note: The block write protection needs to be handled by the caller of the NvM_FirstInit All or by the block owner (which should know about the execution of the NvM_FirstInit All function and related job). This is due to the fact that, upon successful completion of the job of the NvM_FirstInitAll, all selected blocks should have a well known and well defined state.

[SWS_NvM_00932] ⌈The write block once functionality shall not be triggered by NvM_ FirstInitAll.⌋*()*

Note: The reason behind this is to allow the blocks that are Write Once Only to be written via the NvM_WriteBlock API with valid values, after being initialized by the job of the function NvM_FirstInitAll.

[SWS_NvM_00933] ⌈The locked status of a block shall be completely ignored by the NvM_FirstInitAll functionality.⌋*()*

Note: The block locking needs to be handled by the caller of the NvM_FirstInitAll or by the block owner (which should know about the execution of the NvM_FirstInitAll function and related job). This is due to the fact that, upon successful completion of the job of the NvM_FirstInitAll, all selected blocks should have a well known and well defined state.

[SWS_NvM_00934] ⌈If a block that has either permanent RAM or explicit synchronization has been successfully written into the non-volatile memory by the job of the function NvM_FirstInitAll then its RAM block state shall be set to VALID / UNCHANGED.⌋*()*

[SWS_NvM_00935] ⌈If a block that has either permanent RAM or explicit synchronization has been successfully invalidated by the job of the function NvM_FirstInitAll then its RAM block state shall be left untouched.⌋*()*

[SWS_NvM_00936] ⌈The job of the function NvM_FirstInitAll shall not be started while there are single block requests that need to be processed by the NvM module.⌋*()*

[SWS_NvM_00937] ⌈The job of the function NvM_FirstInitAll, once started, shall not be interrupted by any single block requests except write requests for immediate blocks.⌋*()*

[SWS_NvM_00938] ⌈If the NvM module is not initialized and the function NvM_FirstInit All is called, it shall report the Det error NVM_E_UNINIT and return without performing any other activities.⌋*()*

[SWS_NvM_00939] ⌈If a multi block operation is PENDING and the function NvM_ FirstInitAll is called, it shall report the Det error NVM_E_BLOCK_PENDING and return without performing any other activities.⌋*()*

Note: The error NVM_E_BLOCK_PENDING is used to indicate that another multiblock operation is accepted but not completed by NvM. This is due to the fact that the NvM module can only accept and process one multiblock operation at a time.

[SWS_NvM_00940] ⌈The job of the function NvM_FirstInitAll shall set the multi block request result to NVM_REQ_NOT_OK if the processing of at least one NVRAM block fails.⌋*()*

Note: When the result of an individual block processing (in the context of a multi-block job) is different than NVM_REQ_OK, the individual block processing is considered as failed.

8.3.3.6 Callback notifications

[SWS_NvM_00438] ⌈The NvM module shall provide callback functions to be used by the underlying memory abstraction (EEPROM abstraction / FLASH EEPROM Emulation) to signal end of job state with or without error.⌋*()*

8.3.3.6.1 NVRAM Manager job end notification without error

[SWS_NvM_00462] ⌈

Service Name	NvM_JobEndNotification
Syntax	`void NvM_JobEndNotification ( void )`
Service ID [hex]	0x11
Sync/Async	Synchronous
Reentrancy	Non Reentrant
Parameters (in)	None
Parameters (inout)	None
Parameters (out)	None
Return value	None
Description	Function to be used by the underlying memory abstraction to signal end of job without error.
Available via	NvM_MemIf.h

⌋*()*

[SWS_NvM_00111] ⌈The callback function NvM_JobEndNotification is used by the underlying memory abstraction to signal end of job without error.

Note: Successful job end notification of the memory abstraction:

- Read finished & OK
- Write finished & OK
- Erase finished & OK

This routine might be called in interrupt context, depending on the calling function. All memory abstraction modules should be configured to use the same mode (callback/ polling).⌋*()*

[SWS_NvM_00440] ⌈The NvM module shall only provide the callback function NvM_JobEndNotification if polling mode is disabled via NvMPollingMode. The function NvM_JobEndNotification is affected by the common 10.2.2 configuration parameters.⌋*()*

8.3.3.6.2 NVRAM Manager job end notification with error

[SWS_NvM_00463] ⌈

Service Name	NvM_JobErrorNotification
Syntax	`void NvM_JobErrorNotification ( void )`
Service ID [hex]	0x12
Sync/Async	Synchronous
Reentrancy	Non Reentrant
Parameters (in)	None
Parameters (inout)	None
Parameters (out)	None
Return value	None
Description	Function to be used by the underlying memory abstraction to signal end of job with error.
Available via	NvM_MemIf.h

⌋*(SRS_Mem_00125)*

[SWS_NvM_00112] ⌈The callback function NvM_JobErrorNotification is to be used by the underlying memory abstraction to signal end of job with error.

- Note: Unsuccessful job end notification of the memory abstraction:
- Read aborted or failed
- Write aborted or failed
- Erase aborted or failed

This routine might be called in interrupt context, depending on the calling function. All memory abstraction modules should be configured to use the same mode (callback/ polling).

⌋*()*

[SWS_NvM_00441] ⌈The NvM module shall only provide the callback function NvM_ JobErrorNotification if polling mode is disabled via NvMPollingMode. The function Nv M_JoberrorNotification is affected by the common 10.2.2 configuration parameters.⌋*()*

8.4 Scheduled functions

These functions are directly called by Basic Software Scheduler. The following functions shall have no return value and no parameter. All functions shall be non reentrant.

[SWS_NvM_00464] ⌈

Service Name	NvM_MainFunction

Syntax	`void NvM_MainFunction (`
	`  void`
	`)`

Service ID [hex]	0x0e
Description	Service for performing the processing of the NvM jobs.
Available via	SchM_NvM.h

⌋(*SRS_BSW_00425, SRS_BSW_00373, SRS_BSW_00172*)

[SWS_NvM_00256] ⌈The function NvM_MainFunction shall perform the processing of the NvM module jobs.⌋*()*

[SWS_NvM_00333] ⌈The function NvM_MainFunction shall perform the CRC recalculation if requested for a NVRAM block in addition to SWS_NvM_00256.⌋*()*

[SWS_NvM_00334] ⌈The NvM module shall only start writing of a block (i.e. hand over the job to the lower layers) after CRC calculation for this block has been finished.⌋*()*

[SWS_NvM_00257] ⌈The NvM module shall only do/start job processing, queue management and CRC recalculation if the NvM_Init function has internally set an "INIT DONE" signal.⌋*()*

[SWS_NvM_00258] ⌈The function NvM_MainFunction shall restart a destructively canceled request caused by an immediate priority request after the NvM module has processed the immediate priority request [SWS_NvM_00182].⌋*()*

[SWS_NvM_00259] ⌈The function NvM_MainFunction shall supervise the immediate priority queue (if configured) regarding the existence of immediate priority requests.⌋*()*

[SWS_NvM_00346] ⌈If polling mode is enabled, the function NvM_MainFunction shall check the status of the requested job sent to the lower layer.⌋*()*

[SWS_NvM_00347] ⌈If callback routines are configured, the function NvM_MainFunction shall call callback routines to the upper layer after completion of an asynchronous service.⌋*()*

[SWS_NvM_00350] ⌈In case of processing an NvM_WriteAll multi block request, the function NvM_MainFunction shall not call callback routines to the upper layer as long as the service MemIf_GetStatus returns MEMIF_BUSY_INTERNAL for the reserved device ID MEMIF_BROADCAST_ID [6]. For this purpose (status is MEMIF_BUSY_INTERNAL), the function NvM_MainFunction shall cyclically poll the status of the Memory Hardware Abstraction independent of being configured for polling or callback mode.⌋*()*

[SWS_NvM_00349] ⌈The function NvM_MainFunction shall return immediately if no further job processing is possible.⌋*()*

[SWS_NvM_00721] ⌈NVRAM blocks with immediate priority are not expected to be configured to have a CRC.⌋*()*

8.5 Expected interfaces

In this chapter all interfaces required from other modules are listed.

[SWS_NvM_00969] ⌈The NvM shall call the expected interface in the same partition context to which its functionality is mapped:

- The master partition for all general functionality, not directly related to an individual NV block
- The corresponding satellite partition for all functionality directly related to an individual NV block

⌋*()*

8.5.1 Mandatory Interfaces

The following table defines all interfaces which are required to fulfill the core functionality of the module.

[SWS_NvM_00465] ⌈

API Function	*Header File*	*Description*
MemIf_Cancel	MemIf.h	Invokes the "Cancel" function of the underlying memory abstraction module selected by the parameter DeviceIndex.
MemIf_EraseImmediateBlock	MemIf.h	Invokes the "EraseImmediateBlock" function of the underlying memory abstraction module selected by the parameter DeviceIndex.
MemIf_GetJobResult	MemIf.h	Invokes the "GetJobResult" function of the underlying memory abstraction module selected by the parameter DeviceIndex.
MemIf_GetStatus	MemIf.h	Invokes the "GetStatus" function of the underlying memory abstraction module selected by the parameter DeviceIndex.
MemIf_InvalidateBlock	MemIf.h	Invokes the "InvalidateBlock" function of the underlying memory abstraction module selected by the parameter DeviceIndex.
MemIf_Read	MemIf.h	Invokes the "Read" function of the underlying memory abstraction module selected by the parameter DeviceIndex.
MemIf_Write	MemIf.h	Invokes the "Write" function of the underlying memory abstraction module selected by the parameter DeviceIndex.

⌋*(SRS_BSW_00383, SRS_BSW_00384)*

8.5.2 Optional Interfaces

The following table defines all interfaces which are required to fulfill an optional functionality of the module.

[SWS_NvM_00466] ⌈

API Function	*Header File*	*Description*
Crc_CalculateCRC16	Crc.h	This service makes a CRC16 calculation on Crc_ Length data bytes.
Crc_CalculateCRC32	Crc.h	This service makes a CRC32 calculation on Crc_ Length data bytes.
Crc_CalculateCRC8	Crc.h	This service makes a CRC8 calculation on Crc_ Length data bytes, with SAE J1850 parameters
Dem_SetEventStatus	Dem.h	Called by SW-Cs or BSW modules to report monitor status information to the Dem. BSW modules calling Dem_SetEventStatus can safely ignore the return value. This API will be available only if ({Dem/Dem ConfigSet/DemEventParameter/DemEvent ReportingType} == STANDARD_REPORTING)
Det_ReportError	Det.h	Service to report development errors.

⌋*(SRS_BSW_00383, SRS_BSW_00384)*

8.5.3 Configurable interfaces

In this chapter, all interfaces are listed for which the target function can be configured. The target function is usually a callback function. The names of these interfaces are not fixed because they are configurable.

[SWS_NvM_00113] ⌈The notification of a caller via an asynchronous callback routine (NvMSingleBlockCallback) shall be optionally configurable for all NV blocks (see 10.2.3).⌋*()*

[SWS_NvM_00740] ⌈If a callback is configured for a NVRAM block, every asynchronous block request to the block itself shall be terminated with an invocation of the callback routine.⌋*()*

[SWS_NvM_00742] ⌈If no callback is configured for a NVRAM block, there shall be no asynchronous notification of the caller in case of an asynchronous block request.⌋*()*

[SWS_NvM_00260] ⌈A common callback entry (NvMMultiBlockCallback) which is not bound to any NVRAM block shall be optionally configurable for all asynchronous multi block requests (including NvM_CancelWriteAll).⌋*()*

8.5.3.0.1 Single block job end notification

[SWS_NvM_00467] ⌈

Service Name	NvM_SingleBlockCallbackFunction

▽

△

Syntax	`Std_ReturnType NvM_SingleBlockCallbackFunction ( NvM_BlockRequestType BlockRequest, NvM_RequestResultType JobResult )`	
Sync/Async	Synchronous	
Reentrancy	Non Reentrant	
Parameters (in)	BlockRequest	The request type (read, write, ... etc.) of the previous processed block job
	JobResult	The request result of the previous processed block job.
Parameters (inout)	None	
Parameters (out)	None	
Return value	Std_ReturnType	E_OK: callback function has been processed successfully any other: callback function has been processed unsuccessfully
Description	Per block callback routine to notify the upper layer that an asynchronous single block request has been finished.	
Available via	NvM_Externals.h	

⌋(*SRS_BSW_00457*, *SRS_BSW_00360*, *SRS_BSW_00333*)

Note: The following requirements are related to the above mentioned callback SWS_NVM_00176, SWS_NVM_00281, SWS_NvM_00113 and ECUC_NvM_00506.

Note: Please refer to NvMSingleBlockCallback in chapter 10. The Single block job end notification might be called in interrupt context only if there is no callback configured in NvM that belongs to a SW-C.

8.5.3.0.2 Multi block job end notification

[SWS_NvM_00468] ⌈

Service Name	NvM_MultiBlockCallbackFunction	
Syntax	`Std_ReturnType NvM_MultiBlockCallbackFunction ( NvM_MultiBlockRequestType MultiBlockRequest, NvM_RequestResultType JobResult )`	
Sync/Async	Synchronous	
Reentrancy	Non Reentrant	
Parameters (in)	MultiBlockRequest	The request type (read, write, ... etc.) of the previous processed multi block job
	JobResult	The request result of the previous processed multi block job.
Parameters (inout)	None	
Parameters (out)	None	
Return value	Std_ReturnType	E_OK: callback function has been processed successfully any other: callback function has been processed unsuccessfully
Description	Common callback routine to notify the upper layer that an asynchronous multi block request has been finished.	
Available via	NvM_Externals.h	

⌋(*SRS_BSW_00457, SRS_BSW_00360, SRS_BSW_00333*)

Note: The following requirements are related to the above mentioned callback SWS_NVM_00179, SWS_NVM_00260 and ECUC_NvM_00500.

Note: Please refer to NvMMultiBlockCallback in chapter 10. The Multi block job end notification might be called in interrupt context, depending on the calling function.

8.5.3.0.3 Callback function for block initialization

[SWS_NvM_00469] ⌈

Service Name	NvM_InitBlockCallbackFunction	
Syntax	`Std_ReturnType NvM_InitBlockCallbackFunction ( NvM_InitBlockRequestType InitBlockRequest )`	
Sync/Async	Synchronous	
Reentrancy	Non Reentrant	
Parameters (in)	InitBlockRequest	The request type (read, restore, ... etc.) of the currently processed block
Parameters (inout)	None	
Parameters (out)	None	
Return value	Std_ReturnType	E_OK: callback function has been processed successfully any other: callback function has been processed unsuccessfully
Description	Per block callback routine which shall be called by the NvM module when default data needs to be restored in RAM, and a ROM block is not configured.	
Available via	NvM_Externals.h	

⌋(*SRS_BSW_00457, SRS_BSW_00360, SRS_BSW_00333*)

Note: The following requirements are related to the above mentioned callback: SWS_NVM_00085, SWS_NVM_00266, SWS_NvM_00817 and ECUC_NvM_00116.

[SWS_NvM_00369] ⌈If the Init block callback returns a value different than E_OK then the request result shall be set to NVM_REQ_NOT_OK.

Note: The Init block callback is called either if a read request for a block failed in retrieving the data from the non-volatile memory or if explicit default data recovery is requested. Either way, if the Init block callback does not indicate E_OK, the read/restore default operation has failed completely and the request result needs to reflect this.⌋()

Note: Please refer to NvMInitBlockCallback in chapter 10. The init block callback function might be called in interrupt context only if there is no callback configured in NvM that belongs to a SW-C.

[SWS_NvM_00967] ⌈If the block is configured with CalcRamBlockCrc and if the return value for NvMInitBlockCallback is E_OK then NvM shall synchronize the data with the NvM mirror before calculating the CRC over it. (SRS_Mem_08538, SRS_LIBS_08533, SRS_Mem_00016, SRS_Mem_00018)⌋*()*

8.5.3.0.4 Callback function for RAM to NvM copy

[SWS_NvM_00539] ⌈

Service Name	NvM_WriteRamBlockToNvm	
Syntax	`Std_ReturnType NvM_WriteRamBlockToNvm ( void* NvMBuffer )`	
Sync/Async	Synchronous	
Reentrancy	Non Reentrant	
Parameters (in)	None	
Parameters (inout)	None	
Parameters (out)	NvMBuffer	the address of the buffer where the data shall be written to
Return value	Std_ReturnType	E_OK: callback function has been processed successfully any other: callback function has been processed unsuccessfully
Description	Block specific callback routine which shall be called in order to let the application copy data from RAM block to NvM module's mirror.	
Available via	NvM_Externals.h	

⌋*(SRS_BSW_00457)*

[SWS_NvM_00541] ⌈The RAM to NvM copy callback shall be a function pointer.⌋*()*

Note: Please refer to NvMWriteRamBlockToNvCallback in chapter 10.

8.5.3.0.5 Callback function for NvM to RAM copy

[SWS_NvM_00540] ⌈

Service Name	NvM_ReadRamBlockFromNvm	
Syntax	`Std_ReturnType NvM_ReadRamBlockFromNvm ( const void* NvMBuffer )`	
Sync/Async	Synchronous	
Reentrancy	Non Reentrant	
Parameters (in)	NvMBuffer	the address of the buffer where the data can be read from
Parameters (inout)	None	
Parameters (out)	None	
Return value	Std_ReturnType	E_OK: callback function has been processed successfully any other: callback function has been processed unsuccessfully

▽

△

Description	Block specific callback routine which shall be called in order to let the application copy data from NvM module's mirror to RAM block.
Available via	NvM_Externals.h

⌋*(SRS_LIBS_08533, SRS_BSW_00457)*

[SWS_NvM_00542] ⌈The NvM to RAM copy callback shall be a function pointer.⌋*()*

Note: Please refer to NvMReadRamBlockFromNvCallback in chapter 10.

8.6 API Overview

Request Types	Characteristics of Request Types
Type 1: - NvM_SetDataIndex (...) - NvM_GetDataIndex (...) - NvM_SetBlockProtection (...) - NvM_GetErrorStatus(...) - NvM_SetRamBlockStatus(...)	- synchronous request - affects one RAM block - available for all SW-Cs
Type 2: - NvM_ReadBlock(...) - NvM_WriteBlock(...) - NvM_RestoreBlockDefaults(...) - NvM_EraseNvBlock(...) - NvM_InvalidateNvBlock(...) - NvM_CancelJobs(...) - NvM_ReadPRAMBlock(...) - NvM_WritePRAMBlock(...) - NvM_RestorePRAMBlockDefaults(...)	- asynchronous request (result via callback or polling) - affects one NVRAM block - handled by NVRAM manager task via request list - available for all SW-Cs
Type 3: - NvM_ReadAll(...) - NvM_WriteAll(...) - NvM_CancelWriteAll(...) - NvM_ValidateAll(...)	- asynchronous request (result via callback or polling) - affects all NVRAM blocks with permanent RAM data
Type 4: - NvM_Init(...)	- synchronous request - basic initialization - success signaled to the task via command interface inside the function itself

•

8.7 Service Interfaces

This chapter is an addition to the specification of the NvM module. Whereas the other parts of the specification define the behavior and the C-interfaces of the corresponding basic software module, this chapter formally specifies the corresponding AUTOSAR service in terms of the SWC template. The interfaces described here will be visible on the VFB and are used to generate the RTE between application software and the NvM module. [ref. to doc. [13], [14]]

8.7.1 Client-Server-Interfaces

8.7.1.1 NvM_Admin

[SWS_NvM_00737] ⌈

Name	NvMAdmin		
Comment	–		
IsService	true		
Variation	–		
Possible Errors	0	E_OK	Operation successful
	1	E_NOT_OK	Operation failed

Operation	SetBlockProtection	
Comment	Service for setting/resetting the write protection for a NV block.	

Variation	`FOR`
	`  configClass  : ECV.subEltList("NvM/NvMCommon/NvMApiConfigClass");`
	`LET`
	`  isConfigClass3 = configClass.value() == "NVM_API_CONFIG_CLASS_3";`
	`WHERE`
	`  isConfigClass3;`	

Parameters	ProtectionEnabled	
	Type	boolean
	Direction	IN
	Comment	–
	Variation	–
Possible Errors	E_OK E_NOT_OK	

⌋()

8.7.1.2 NvM_Mirror

[SWS_NvM_00738] ⌈

Name	NvMMirror		
Comment	–		
IsService	true		
Variation	–		
Possible Errors	0	E_OK	Operation successful
	1	E_NOT_OK	Operation failed

Operation	ReadRamBlockFromNvM	
Comment	Block specific callback routine which shall be called in order to let the application copy data from NvM module's mirror to RAM block.	
Variation	–	
Parameters	SrcPtr	
	Type	ConstVoidPtr
	Direction	IN
	Comment	The parameter "SrcPtr" shall be typed by an ImplementationDataType of category DATA_REFERENCE with the pointer target void to pass an address (pointer) to the RAM Block.
	Variation	–
Possible Errors	E_OK E_NOT_OK	

Operation	WriteRamBlockToNvM	
Comment	Block specific callback routine which shall be called in order to let the application copy data from RAM block to NvM module's mirror.	
Variation	–	
Parameters	DstPtr	
	Type	VoidPtr
	Direction	IN
	Comment	The parameter "DstPtr" shall be typed by an ImplementationDataType of category DATA_REFERENCE with the pointer target void to pass an address (pointer) to the RAM Block.
	Variation	–
Possible Errors	E_OK E_NOT_OK	

⌋()

8.7.1.3 NvM_NotifyInitBlock

[SWS_NvM_00736] ⌈

Name	NvMNotifyInitBlock		
Comment	Callback that is called by the NvM module when default data needs to be restored to the RAM image		
IsService	true		
Variation	–		
Possible Errors	0	E_OK	RAM block content was updated
	1	RTE_E_RAM_ UNCHANGED	RAM block content was not changed

Operation	InitBlock	
Comment	This callback is called if the initialization of a block has completed.	
Variation	–	
Parameters	InitBlockRequest	
	Type	NvM_InitBlockRequestType
	Direction	IN
	Comment	–
	Variation	–
Possible Errors	–	

⌋()

8.7.1.4 NvM_NotifyJobFinished

[SWS_NvM_00735] ⌈

Name	NvMNotifyJobFinished		
Comment	Callback that is called when a job has finished		
IsService	true		
Variation	–		
Possible Errors	0	E_OK	Operation successful

Operation	JobFinished	
Comment	Callback that gets called if a job has finished	
Variation	–	
Parameters	BlockRequest	
	Type	NvM_BlockRequestType
	Direction	IN
	Comment	–
	Variation	–
	JobResult	
	Type	NvM_RequestResultType
	Direction	IN
	Comment	–
	Variation	–
Possible Errors	E_OK	

⌋()

8.7.1.5 NvM_Service

[SWS_NvM_00734] ⌈

Name	NvMService		
Comment	–		
IsService	true		
Variation	–		
Possible Errors	0	E_OK	Operation successful
	1	E_NOT_OK	Operation failed

Operation	EraseBlock
Comment	Service to erase a NV block.

Variation	`FOR`
	`  configClass  : ECV.subEltList("NvM/NvMCommon/NvMApiConfigClass");`
	`LET`
	`  isConfigClass3 = configClass.value() == "NVM_API_CONFIG_CLASS_3";`
	`WHERE`
	`  isConfigClass3;`

Possible Errors	E_OK E_NOT_OK

Operation	GetDataIndex	
Comment	Service for getting the currently set DataIndex of a dataset NVRAM block	

Variation	`FOR`
	`  configClass  : ECV.subEltList("NvM/NvMCommon/NvMApiConfigClass");`
	`LET`
	`  isConfigClass2 = configClass.value() == "NVM_API_CONFIG_CLASS_2";`
	`WHERE`
	`  isConfigClass2;`	

Parameters	DataIndex	
	Type	uint8
	Direction	OUT
	Comment	–
	Variation	–
Possible Errors	E_OK E_NOT_OK	

Operation	GetErrorStatus	
Comment	Service to read the block dependent error/status information.	
Variation	–	
Parameters	RequestResult	
	Type	NvM_RequestResultType
	Direction	OUT
	Comment	–
	Variation	–
Possible Errors	E_OK E_NOT_OK	

Operation	InvalidateNvBlock
Comment	Service to invalidate a NV block.

▽

△

Variation	**FOR**
	  configClass : ECV.*subEltList*("NvM/NvMCommon/NvMApiConfigClass");
	LET
	  isConfigClass3 = configClass.*value*() == "NVM_API_CONFIG_CLASS_3";
	WHERE
	  isConfigClass3;

Possible Errors	E_OK E_NOT_OK

Operation	ReadBlock	
Comment	Service to copy the data of the NV block to its corresponding RAM block.	

Variation	**FOR**
	  configClass : ECV.*subEltList*("NvM/NvMCommon/NvMApiConfigClass");
	LET
	  isConfigClass2 = configClass.*value*() == "NVM_API_CONFIG_CLASS_2";
	  isConfigClass3 = configClass.*value*() == "NVM_API_CONFIG_CLASS_3";
	WHERE
	  isConfigClass2 **OR** isConfigClass3;	

Parameters	DstPtr	
	Type	VoidPtr
	Direction	IN
	Comment	The parameter "DstPtr" shall be typed by an ImplementationDataType of category DATA_REFERENCE with the pointer target void to pass an address (pointer) to the RAM Block.
	Variation	–
Possible Errors	E_OK E_NOT_OK	

Operation	ReadPRAMBlock	
Comment	–	

Variation	**FOR**
	  configClass : ECV.*subEltList*("NvM/NvMCommon/NvMApiConfigClass");
	LET
	  isConfigClass2 = configClass.*value*() == "NVM_API_CONFIG_CLASS_2";
	  isConfigClass3 = configClass.*value*() == "NVM_API_CONFIG_CLASS_3";
	WHERE
	  isConfigClass2 **OR** isConfigClass3;	

Possible Errors	E_OK E_NOT_OK	

Operation	RestoreBlockDefaults	
Comment	Service to restore the default data to its corresponding RAM block.	

Variation	**FOR**
	  configClass : ECV.*subEltList*("NvM/NvMCommon/NvMApiConfigClass");
	LET
	  isConfigClass2 = configClass.*value*() == "NVM_API_CONFIG_CLASS_2";
	  isConfigClass3 = configClass.*value*() == "NVM_API_CONFIG_CLASS_3";
	WHERE
	  isConfigClass2 **OR** isConfigClass3;	

Parameters	DstPtr	
	Type	VoidPtr
	Direction	IN

▽

△

	Comment	The parameter "DstPtr" shall be typed by an ImplementationDataType of category DATA_REFERENCE with the pointer target void to pass an address (pointer) to the RAM Block.
	Variation	–
Possible Errors	E_OK E_NOT_OK	

Operation	RestorePRAMBlockDefaults
Comment	–

Variation	`FOR`
	`  configClass  : ECV.subEltList("NvM/NvMCommon/NvMApiConfigClass");`
	`LET`
	`  isConfigClass2 = configClass.value() == "NVM_API_CONFIG_CLASS_2";`
	`  isConfigClass3 = configClass.value() == "NVM_API_CONFIG_CLASS_3";`
	`WHERE`
	`  isConfigClass2 OR isConfigClass3;`

Possible Errors	E_OK E_NOT_OK

Operation	SetDataIndex	
Comment	Service for setting the DataIndex of a dataset NVRAM block.	

Variation	`FOR`
	`  configClass  : ECV.subEltList("NvM/NvMCommon/NvMApiConfigClass");`
	`LET`
	`  isConfigClass2  = configClass.value() == "NVM_API_CONFIG_CLASS_2";`
	`  isConfigClass3  = configClass.value() == "NVM_API_CONFIG_CLASS_3";`
	`  blockMgmTypes  = ECV.subEltList("NvM/NvMBlockDescriptor/`
	`  NvMBlockManagementType");`
	`  isMgd(mgmtType) = mgmtType.value() == "NVM_BLOCK_DATASET";`
	`  datasetMgdCount = blockMgmTypes.filter(isMgd).count();`
	`WHERE`
	`  (isConfigClass2 OR isConfigClass3) AND (datasetMgdCount GT 0);`	

Parameters	DataIndex	
	Type	uint8
	Direction	IN
	Comment	–
	Variation	–
Possible Errors	E_OK E_NOT_OK	

Operation	SetRamBlockStatus	
Comment	Service for setting the RAM block status of an NVRAM block.	

Variation	`LET`
	`  nvmBlockUseSetRamBlockStatus =  ECV.subEltList("NvM/`
	`  NvMBlockDescriptor/NvMBlockUseSetRamBlockStatus");`
	`  useSetRamBlockStatus(useApi) =  useApi.value() == true;`
	`  useSetRamBlockStatusCount  =  nvmBlockUseSetRamBlockStatus.filter(`
	`  useSetRamBlockStatus).count();`
	`WHERE`
	`  (useSetRamBlockStatusCount GT 0);`	

Parameters	BlockChanged	
	Type	boolean
	Direction	IN

▽

△

	Comment	–
	Variation	–
Possible Errors	E_OK E_NOT_OK	

Operation	WriteBlock	
Comment	Service to copy the data of the RAM block to its corresponding NV block.	

Variation	**FOR**
	  configClass : ECV.*subEltList*("NvM/NvMCommon/NvMApiConfigClass");
	LET
	  isConfigClass2 = configClass.*value*() == "NVM_API_CONFIG_CLASS_2";
	  isConfigClass3 = configClass.*value*() == "NVM_API_CONFIG_CLASS_3";
	WHERE
	  isConfigClass2 **OR** isConfigClass3;	

Parameters	SrcPtr	
	Type	ConstVoidPtr
	Direction	IN
	Comment	The parameter "SrcPtr" shall be typed by an ImplementationDataType of category DATA_REFERENCE with the pointer target void to pass an address (pointer) to the RAM Block.
	Variation	–
Possible Errors	E_OK E_NOT_OK	

Operation	WritePRAMBlock	
Comment	–	

Variation	**FOR**
	  configClass : ECV.*subEltList*("NvM/NvMCommon/NvMApiConfigClass");
	LET
	  isConfigClass2 = configClass.*value*() == "NVM_API_CONFIG_CLASS_2";
	  isConfigClass3 = configClass.*value*() == "NVM_API_CONFIG_CLASS_3";
	WHERE
	  isConfigClass2 **OR** isConfigClass3;	

Possible Errors	E_OK E_NOT_OK	

⌋()

8.7.2 Implementation Data Types

Note: The header Rte_NvM_Type.h is only available in case there is a SwComponent Type with ShortName or component type symbol 'NvM' (see [SWS_Rte_07122]). This SwComponent has to use the needed types (see [SWS_Rte_08802]). This means that especially in systems where only NvBlockSwComponents are present, there needs to be an additional, minimalistic SwComponentType named NvM referencing the required data types by an IncludedDataTypeSet.

8.7.2.1 ImplementationDataType NvM_RequestResultType

[SWS_NvM_00470] ⌈

Name	NvM_RequestResultType		
Kind	Type		
Derived from	uint8		
Range	NVM_REQ_OK	0x00	The last asynchronous request has been finished successfully. This shall be the default value after reset. This status shall have the value 0.
	NVM_REQ_NOT_OK	0x01	The last asynchronous read/write/ control request has been finished unsuccessfully.
	NVM_REQ_PENDING	0x02	An asynchronous read/write/ control request is currently pending.
	NVM_REQ_INTEGRITY_FAILED	0x03	The result of the last asynchronous request NvM_ReadBlock or NvM_ReadAll is a data integrity failure. Note: In case of NvM_ReadBlock the content of the RAM block has changed but has become invalid. The application is responsible to renew and validate the RAM block content.
	NVM_REQ_BLOCK_SKIPPED	0x04	The referenced block was skipped during execution of NvM_ReadAll or NvM_WriteAll, e.g. Dataset NVRAM blocks (NvM_ReadAll) or NVRAM blocks without a permanently configured RAM block.
	NVM_REQ_NV_INVALIDATED	0x05	The referenced NV block is invalidated.
	NVM_REQ_CANCELED	0x06	The multi block request NvM_ WriteAll was canceled by calling NvM_CancelWriteAll. Or Any single block job request (NvM_ ReadBlock, NvM_WriteBlock, Nv M_EraseNvBlock, NvM_Invalidate NvBlock and NvM_RestoreBlock Defaults) was canceled by calling NvM_CancelJobs.
	NVM_REQ_RESTORED_ DEFAULTS	0x08	The referenced NV block had the default values copied to the RAM image.
Description	This is an asynchronous request result returned by the API service NvM_GetErrorStatus. The availability of an asynchronous request result can be additionally signaled via a callback function.		
Variation	–		
Available via	Rte_NvM_Type.h		

⌋()

8.7.2.2 ImplementationDataType NvM_BlockIdType

[SWS_NvM_00471] ⌈

Name	NvM_BlockIdType		
Kind	Type		
Derived from	uint16		
Range	0..2^(16- NvMDatasetSelection Bits)-1	–	–
Description	Identification of a NVRAM block via a unique block identifier. Reserved NVRAM block IDs: 0 -> to derive multi block request results via NvM_GetErrorStatus 1 -> redundant NVRAM block which holds the configuration ID		
Variation	–		
Available via	Rte_NvM_Type.h		

⌋()

8.7.2.3 ImplementationDataType NvM_InitBlockRequestType

[SWS_NvM_91123] ⌈

Name	NvM_InitBlockRequestType		
Kind	Type		
Derived from	uint8		
Range	NVM_INIT_READ_BLOCK	0x00	NvM_ReadBlock/ NvM_Read PRAMBlock is requested on the block
	NVM_INIT_RESTORE_BLOCK_ DEFAULTS	0x01	NvM_RestoreBlockDefaults/ Nv M_RestorePRAMBlockDefaults is requested on the block
	NVM_INIT_READ_ALL_BLOCK	0x02	NvM_ReadAll is processing this block
	NVM_INIT_FIRST_INIT_ALL	0x03	NvM_FirstInitAll is processing this block
Description	Identifies the type of request performed on a block when signaled via the callback function		
Variation	–		
Available via	Rte_NvM_Type.h		

⌋()

8.7.2.4 ImplementationDataType NvM_BlockRequestType

[SWS_NvM_91002] ⌈

Name	NvM_BlockRequestType		
Kind	Type		
Derived from	uint8		
Range	NVM_READ_BLOCK	0x00	NvM_ReadBlock/ NvM_Read PRAMBlock was performed on the block
	NVM_WRITE_BLOCK	0x01	NvM_WriteBlock/ NvM_Write PRAMBlock was performed on the block
	NVM_RESTORE_BLOCK_ DEFAULTS	0x02	NvM_RestoreBlockDefaults/ Nv M_RestorePRAMBlockDefaults was performed on the block
	NVM_ERASE_NV_BLOCK	0x03	NvM_EraseNvBlock was performed on the block
	NVM_INVALIDATE_NV_BLOCK	0x04	NvM_InvalidateNvBlock was performed on the block
	NVM_READ_ALL_BLOCK	0x05	NvM_ReadAll has finished processing this block
Description	Identifies the type of request performed on a block when signaled via the callback function		
Variation	–		
Available via	Rte_NvM_Type.h		

⌋() Note: Calling the single block callback with NVM_READ_ALL_BLOCK can be used to trigger an RTE Event that initializes a SW-C (see note below SWS_NvM_00281) as opposed to calling the single block callback with NVM_READ_BLOCK which is used to notify an already initialized SW-C of the result of a pending read block job. Therefore separate literals/values are specified.

8.7.3 Ports

8.7.3.1 NvM_PAdmin_{Block}

[SWS_NvM_00843] ⌈

Name	PAdmin_{Block}		
Kind	ProvidedPort	***Interface***	NvMAdmin
Description	–		
Port Defined Argument Value(s)	***Type***	NvM_BlockIdType	
	Value	`FOR nvBlockDescriptor : ECV.subEltList("NvM/NvMBlockDescriptor"); LET Block = nvBlockDescriptor.shortname(); BlockId = nvBlockDescriptor.subElt("NvMNvramBlockIdentifier").value();`	

▽

△

Variation	**FOR**
	  nvBlockDescriptor : ECV.*subEltList*("NvM/NvMBlockDescriptor");
	LET
	  Block = nvBlockDescriptor.*shortname*();
	  UsePort = nvBlockDescriptor.*subElt*("NvMBlockUsePort").*value*() == true;
	WHERE
	  UsePort;

⌋()

8.7.3.2 NvM_PM_{Block}

[SWS_NvM_00844] ⌈

Name	PM_{Block}		
Kind	RequiredPort	*Interface*	NvMMirror
Description	–		

Variation	**FOR**
	  nvBlockDescriptor : ECV.*subEltList*("NvM/NvMBlockDescriptor");
	LET
	  Block = nvBlockDescriptor.*shortname*();
	  UsePort = nvBlockDescriptor.*subElt*("NvMBlockUsePort").*value*()
	  == true;
	  UsePortSyncMech = nvBlockDescriptor.*subElt*("NvMBlockUseSyncMechanism")
	  .*value*() == true;
	WHERE
	  UsePort **AND** UsePortSyncMech;		

⌋()

8.7.3.3 NvM_PNIB_{Block}

[SWS_NvM_00845] ⌈

Name	PNIB_{Block}		
Kind	RequiredPort	*Interface*	NvMNotifyInitBlock
Description	–		

▽

△

Variation	`FOR`
	`  nvBlockDescriptor : ECV.subEltList("NvM/NvMBlockDescriptor");`
	`LET`
	`  Block  = nvBlockDescriptor.shortname();`
	`  UsePort  = nvBlockDescriptor.subElt("NvMBlockUsePort").`
	`  value() == true;`
	`  InitBlockCallbackDef  = nvBlockDescriptor.subElt("`
	`  NvMInitBlockCallback").isDefined();`
	`  InitBlockCallbackFncDef = nvBlockDescriptor.subElt("`
	`  NvMInitBlockCallback/NvMInitBlockCallbackFnc").isDefined();`
	`WHERE`
	`  UsePort AND InitBlockCallbackDef AND NOT InitBlockCallbackFncDef;`

⌋()

8.7.3.4 NvM_PNJF_{Block}

[SWS_NvM_00846] ⌈

Name	PNJF_{Block}		
Kind	RequiredPort	***Interface***	NvMNotifyJobFinished
Description	–		

Variation	`FOR`
	`  nvBlockDescriptor : ECV.subEltList("NvM/NvMBlockDescriptor");`
	`LET`
	`  Block  = nvBlockDescriptor.shortname();`
	`  UsePort  = nvBlockDescriptor.subElt("NvMBlockUsePort").`
	`  value() == true;`
	`  SingleBlockCallbackDef  = nvBlockDescriptor.subElt("`
	`  NvMSingleBlockCallback").isDefined();`
	`  SingleBlockCallbackFncDef = nvBlockDescriptor.subElt("`
	`  NvMSingleBlockCallback/NvMSingleBlockCallbackFnc").isDefined();`
	`WHERE`
	`  UsePort AND SingleBlockCallbackDef AND NOT SingleBlockCallbackFncDef;`		

⌋()

8.7.3.5 NvM_PS_{Block}

[SWS_NvM_00847] ⌈

Name	PS_{Block}		
Kind	ProvidedPort	***Interface***	NvMService
Description	–		
Port Defined Argument Value(s)	***Type***	NvM_BlockIdType	

▽

△

	Value	**FOR**
		  nvBlockDescriptor : ECV.*subEltList*("NvM/
		  NvMBlockDescriptor");
		LET
		  Block = nvBlockDescriptor.*shortname*();
		  BlockId = nvBlockDescriptor.*subElt*("
		  NvMNvramBlockIdentifier").*value*();

Variation	**FOR**
	  nvBlockDescriptor : ECV.*subEltList*("NvM/NvMBlockDescriptor");
	LET
	  Block = nvBlockDescriptor.*shortname*();
	  UsePort = nvBlockDescriptor.*subElt*("NvMBlockUsePort").*value*() == true;
	WHERE
	  UsePort;	

⌋()

9 Sequence diagrams

9.1 Synchronous calls

9.1.1 NvM_Init

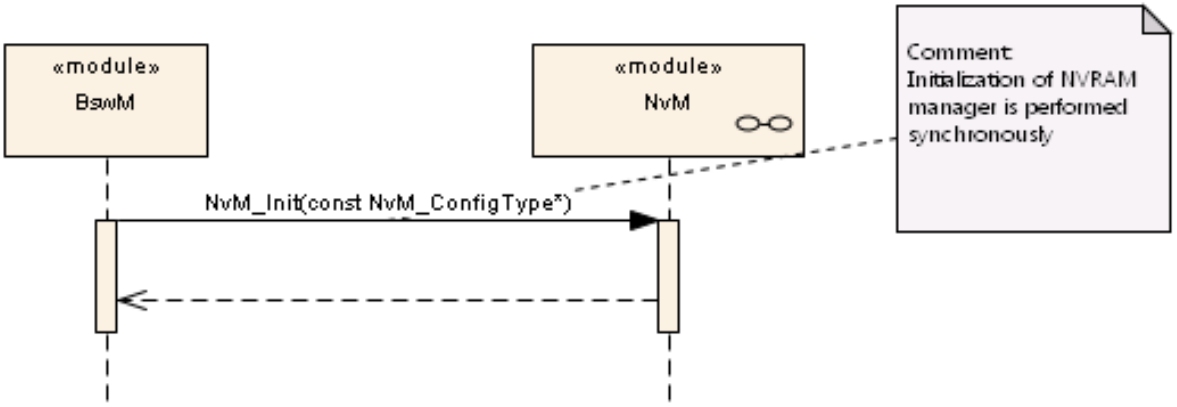

Figure 9.1: UML sequence diagram NvM_Init

9.1.2 NvM_SetDataIndex

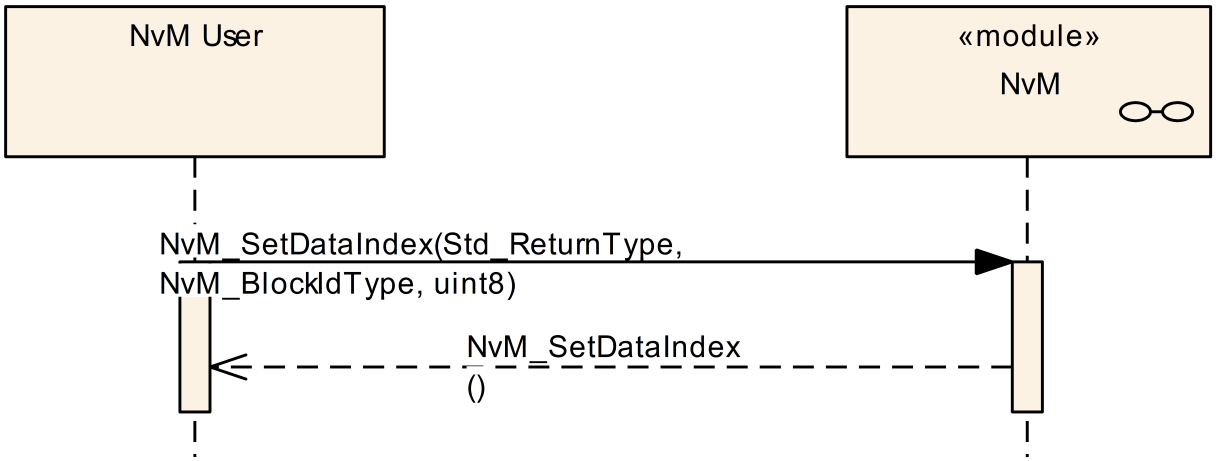

Figure 9.2: UML sequence diagram NvM_SetDataIndex

9.1.3 NvM_GetDataIndex

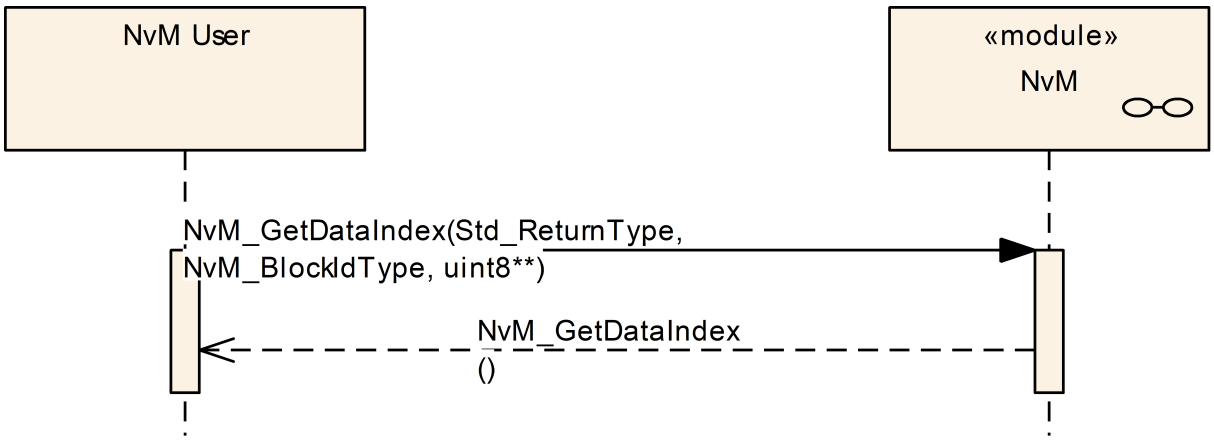

Figure 9.3: UML sequence diagram NvM_GetDataIndex

9.1.4 NvM_SetBlockProtection

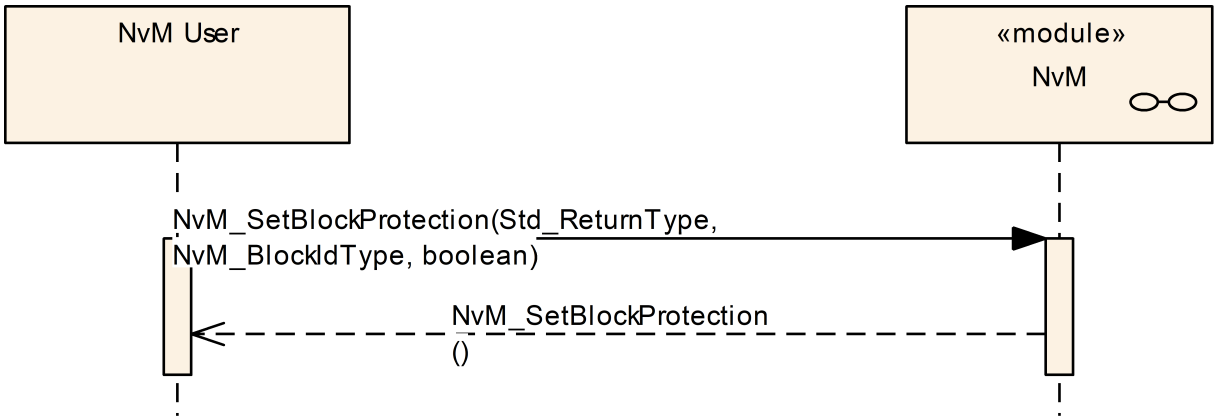

Figure 9.4: UML sequence diagram NvM_SetBlockProtection

9.1.5 NvM_GetErrorStatus

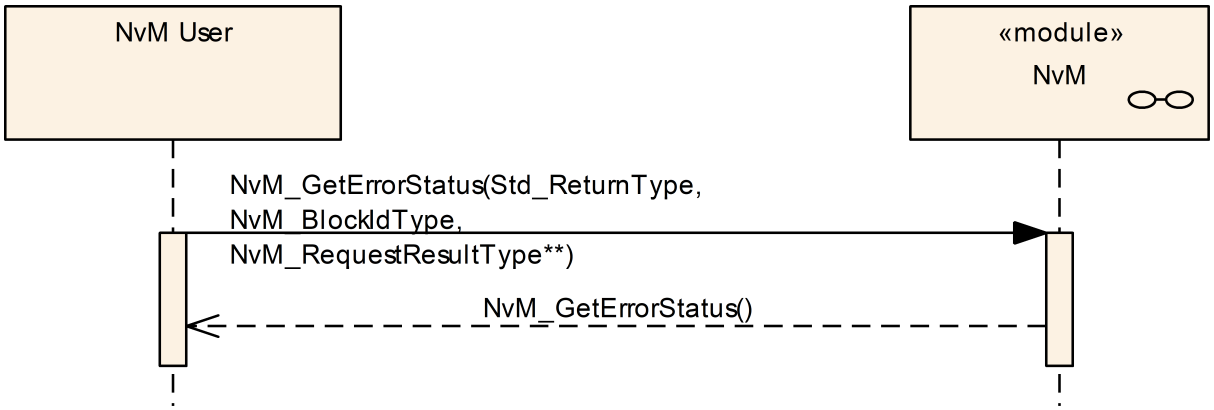

Figure 9.5: UML sequence diagram NvM_GetErrorStatus

9.1.6 NvM_GetVersionInfo

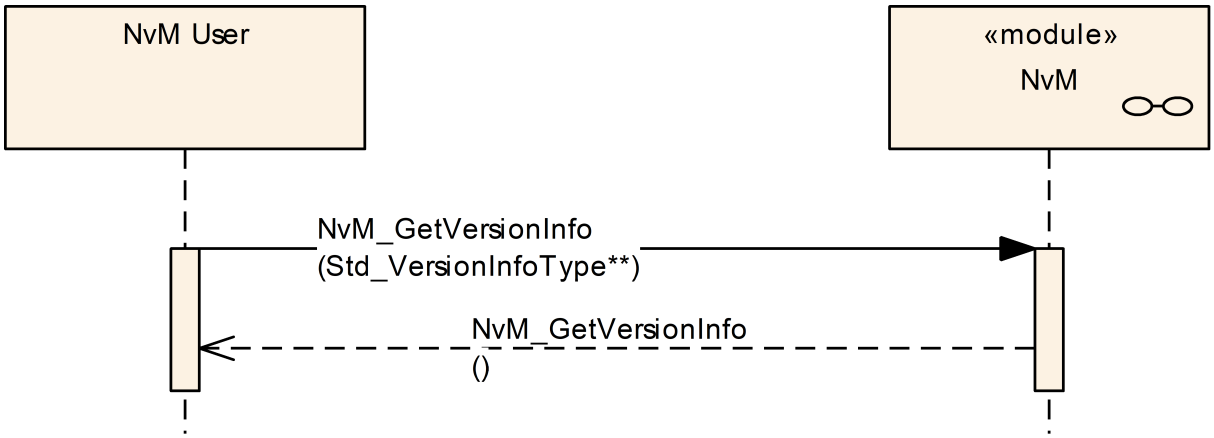

Figure 9.6: UML sequence diagram NvM_GetVersionInfo

9.2 Asynchronous calls

The following sequence diagrams concentrate on the interaction between the NvM module and SW-C's or the BSW Mode Manager. For interaction regarding the Memory Interface please ref. to [4] or [5].

9.2.1 Asynchronous call with polling

The following diagram shows the function NvM_WriteBlock as an example of a request that is performed asynchronously. The sequence for all other asynchronous functions is the same, only the processed number of blocks and the block types may vary. The result of the asynchronous function is obtained by polling requests to the error/status information.

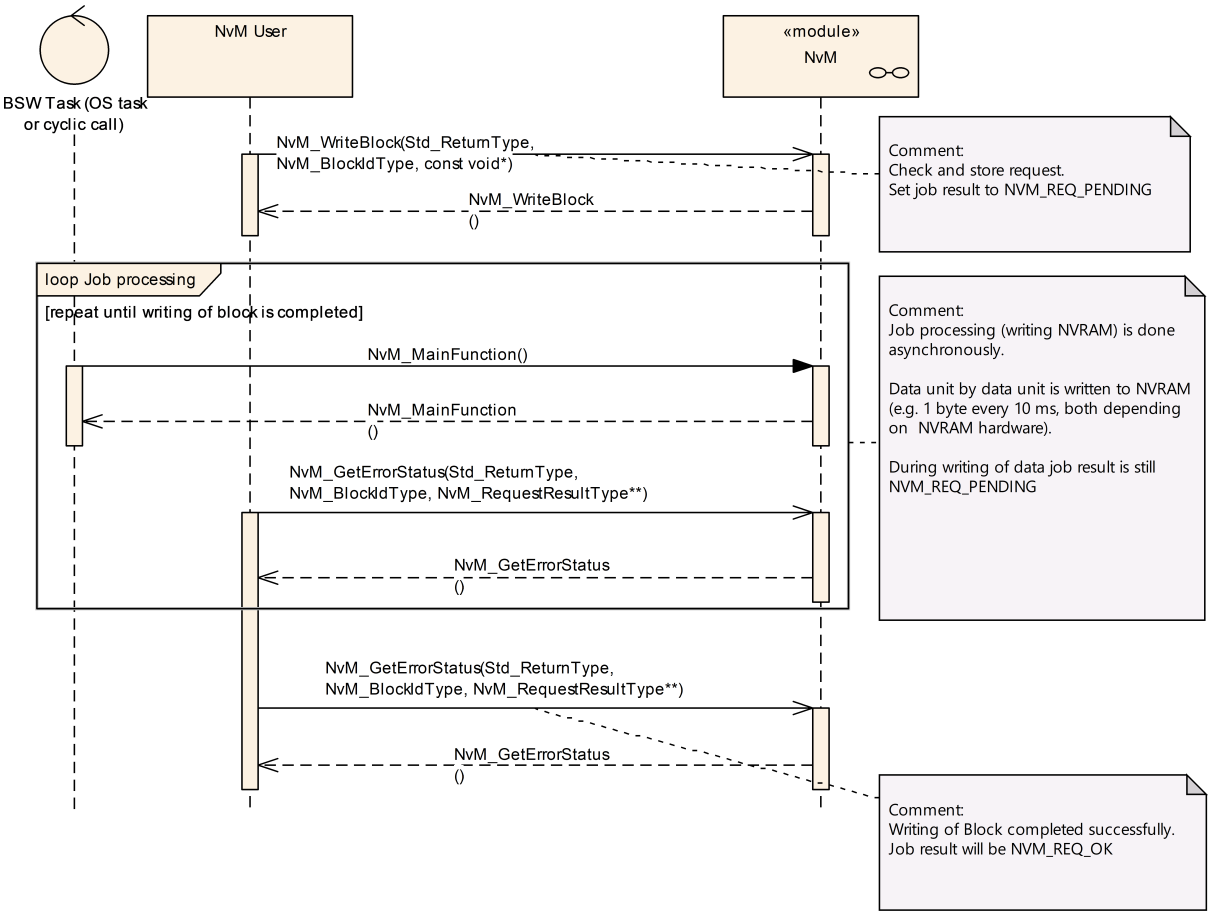

Figure 9.7: UML sequence diagram for asynchronous call with polling

9.2.2 Asynchronous call with callback

The following diagram shows the function NvM_WriteBlock as an example of a request that is performed asynchronously. The sequence for all other asynchronous functions is the same, only the processed number of blocks and the block types may vary. The result of the asynchronous function is obtained after an asynchronous notification (call-back) by requesting the error/status information.

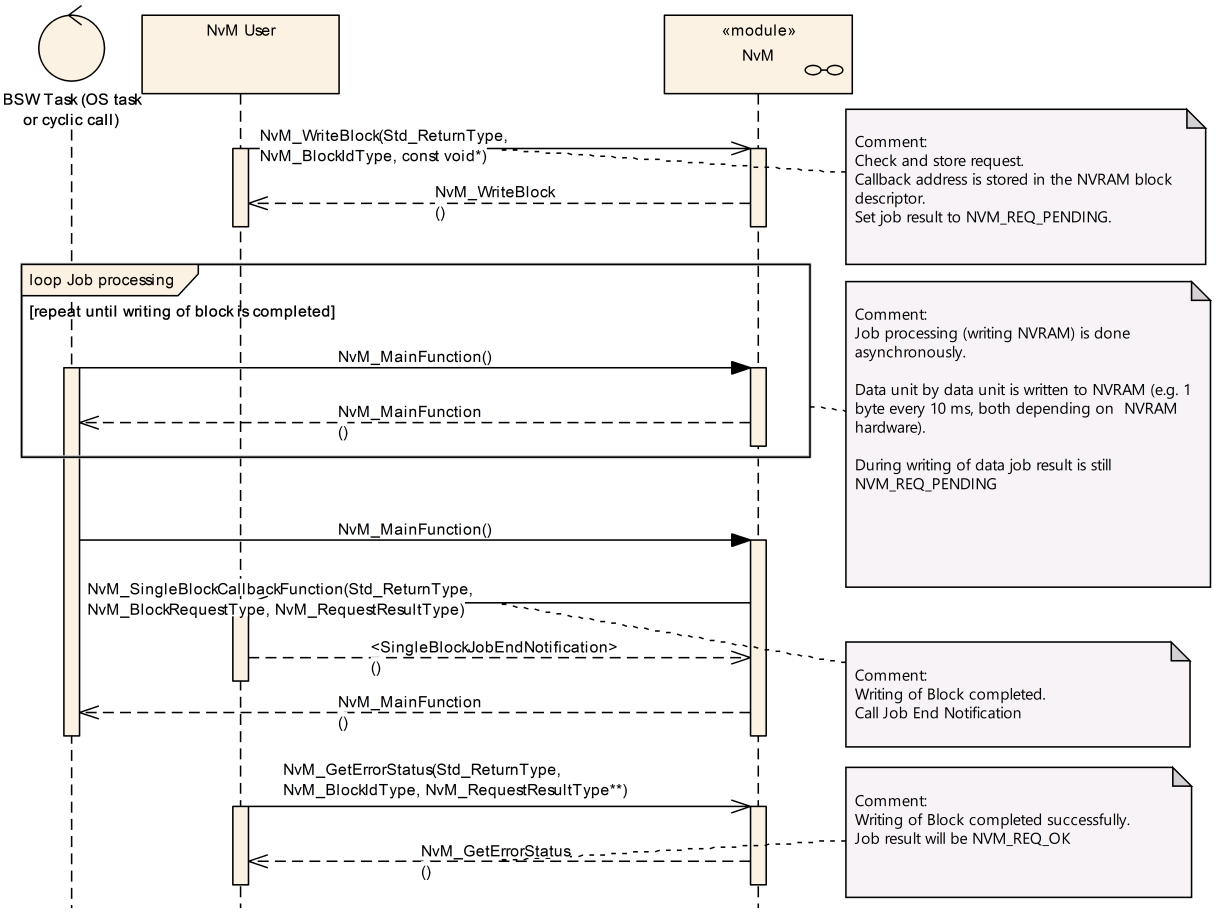

Figure 9.8: UML sequence diagram for asynchronous call with callback

9.2.3 Cancellation of a Multi Block Request

The following diagram shows the effect of a cancel operation applied to a running NvM_WriteAll multi block request. The running NvM_WriteAll function completes the actual NVRAM block and stops further writes.

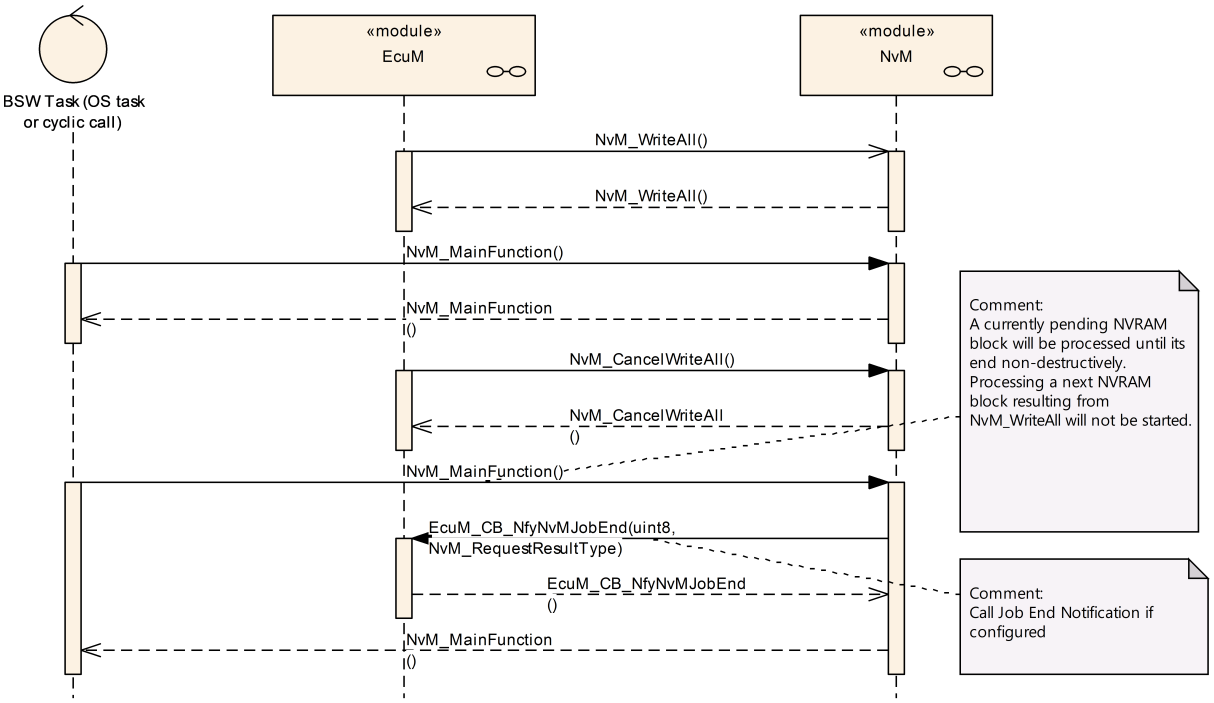

Figure 9.9: UML sequence diagram for cancellation of asynchronous call

9.2.4 BswM Interraction

The following diagrams show the interractions between NvM and BswM in terms of single block operation and multiblock operation.

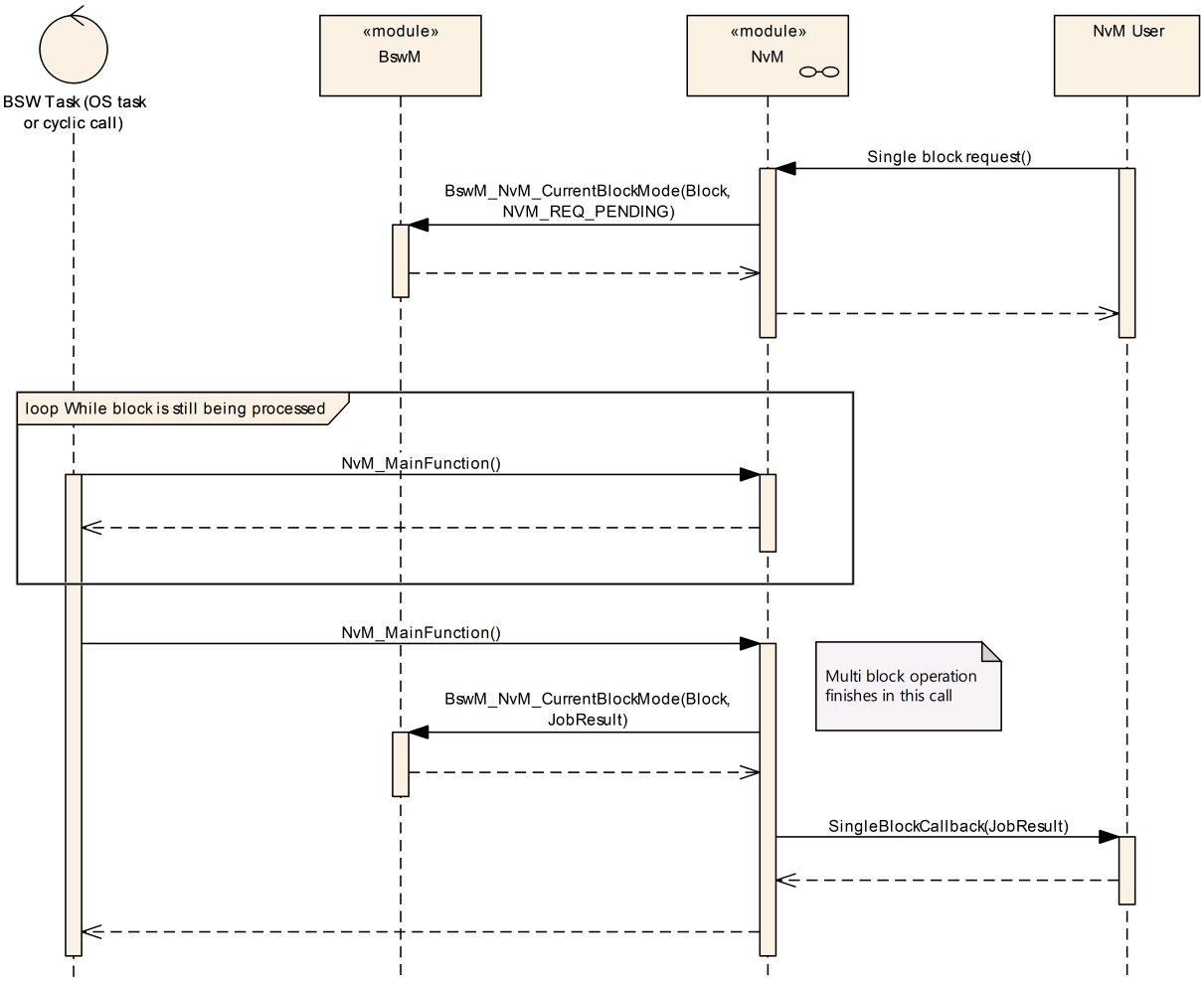

Figure 9.10: NvM interraction with BswM in case of a single block operation

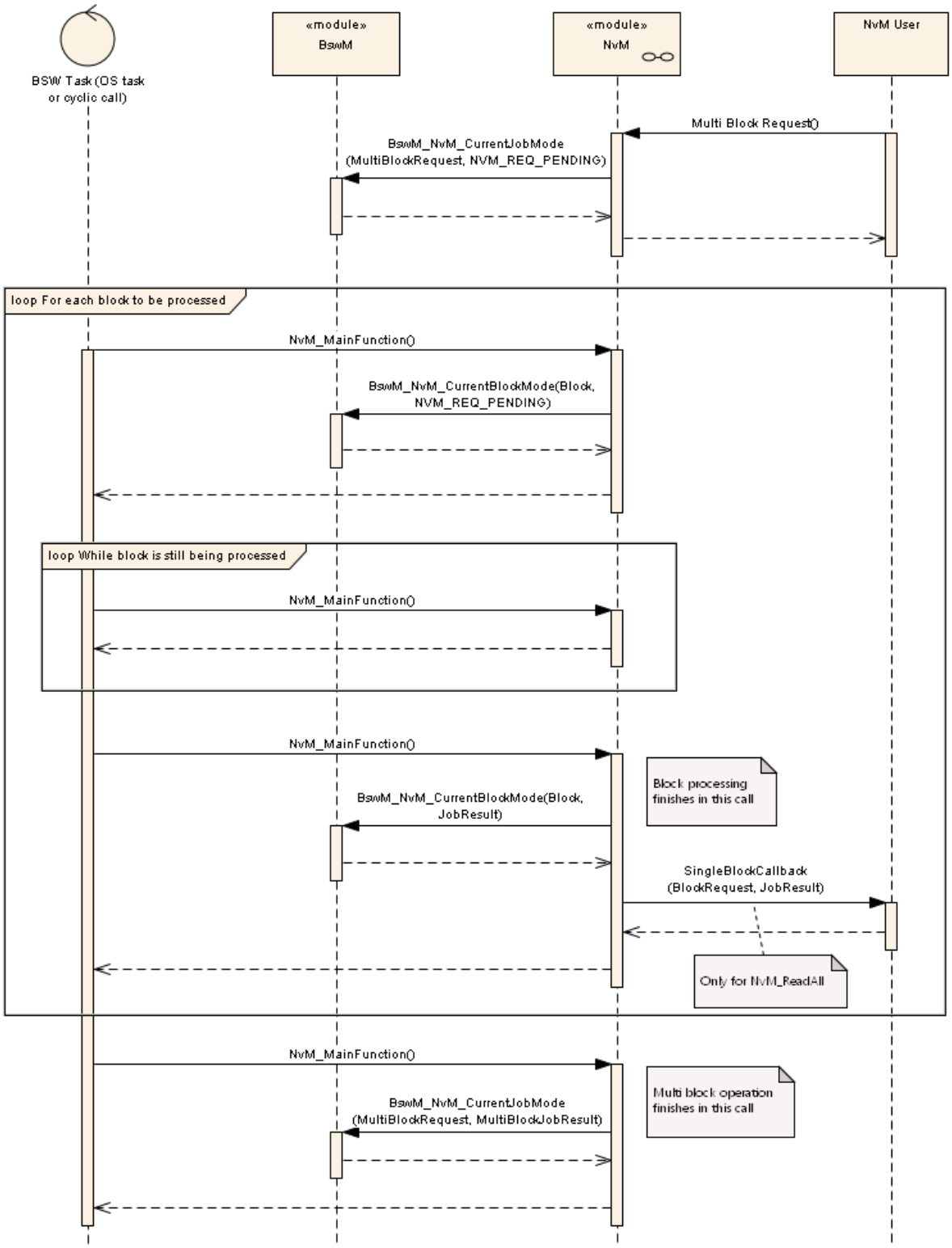

Figure 9.11: NvM interraction with BswM for a multiblock operation

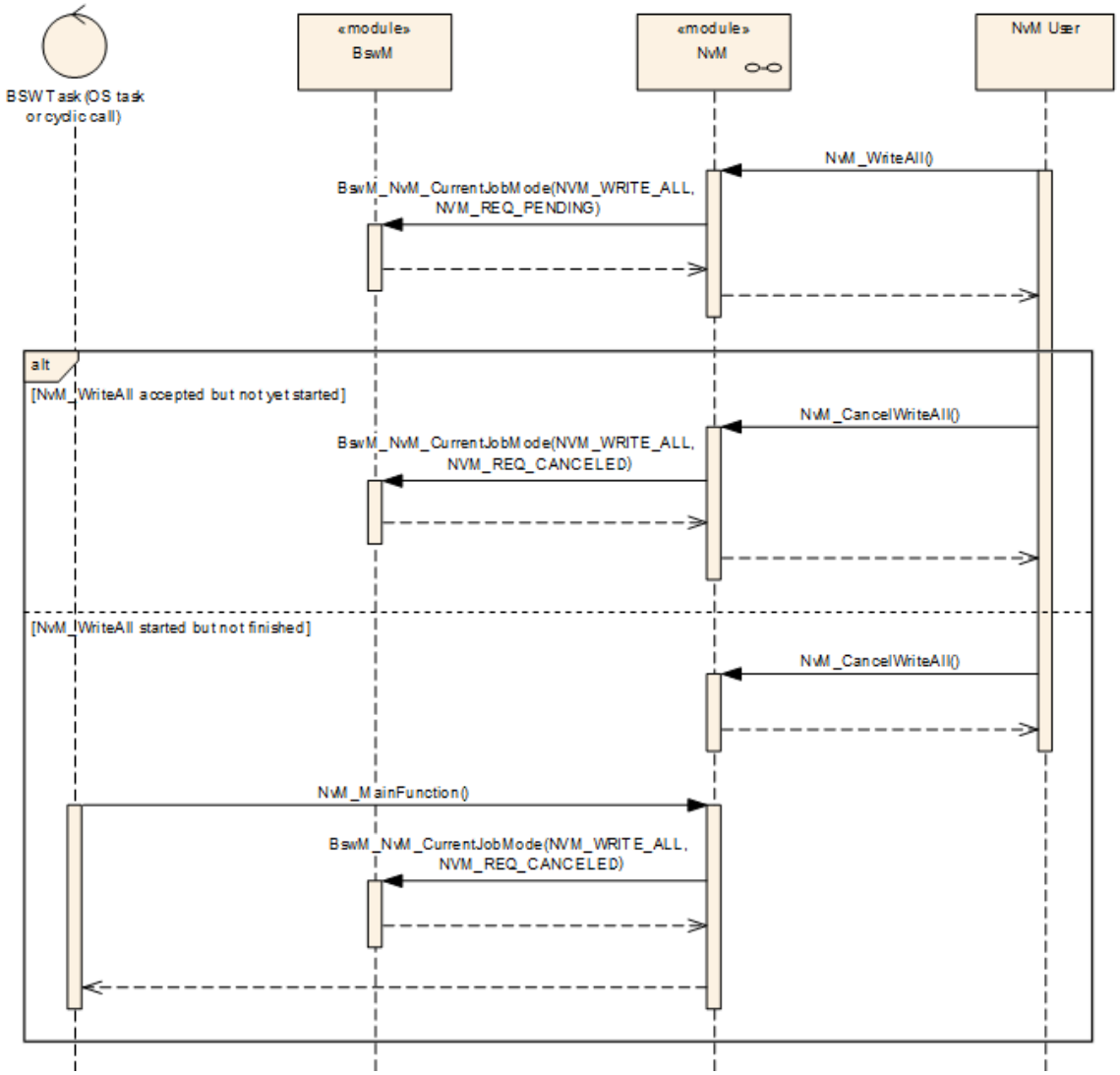

Figure 9.12: NvM interaction with BswM in case of a WriteAll cancellation

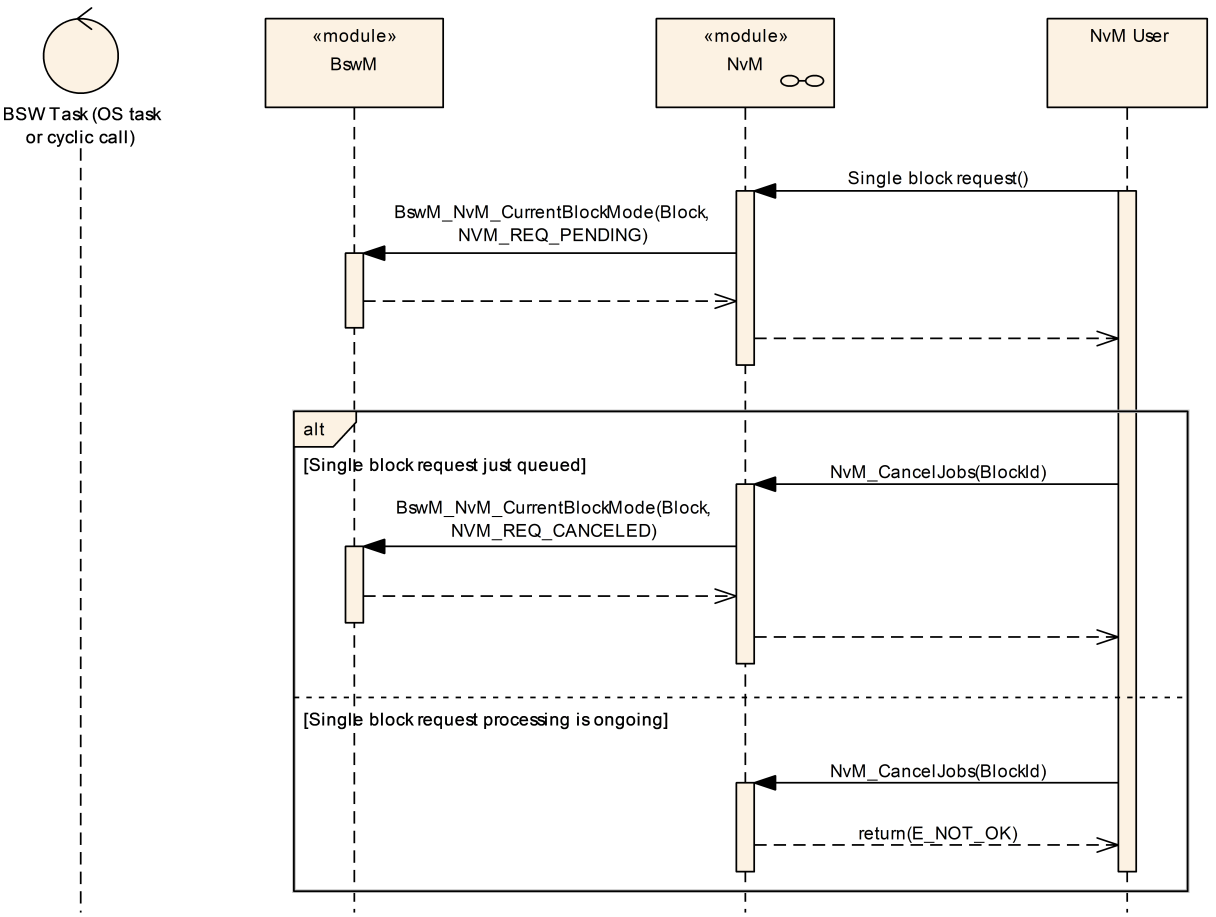

Figure 9.13: NvM interraction with BswM in case of a single block cancellation

10 Configuration specification

In general, this chapter defines configuration parameters and their clustering into containers. In order to support the specification Chapter 10.1 describes fundamentals. It also specifies a template (table) you shall use for the parameter specification. We intend to leave Chapter 10.1 in the specification to guarantee comprehension.

Chapter 10.2 specifies the structure (containers) and the parameters of the module NvM.

Chapter 10.3 specifies published information of the module NvM.

10.1 How to read this chapter

For details refer to the chapter 10.1 “Introduction to configuration specification” in SWS_BSWGeneral. [ref. to doc. [2]]

10.2 Containers and configuration parameters

The following chapters summarize all configuration parameters. The detailed meanings of the parameters describe Chapter 7 and Chapter 8.

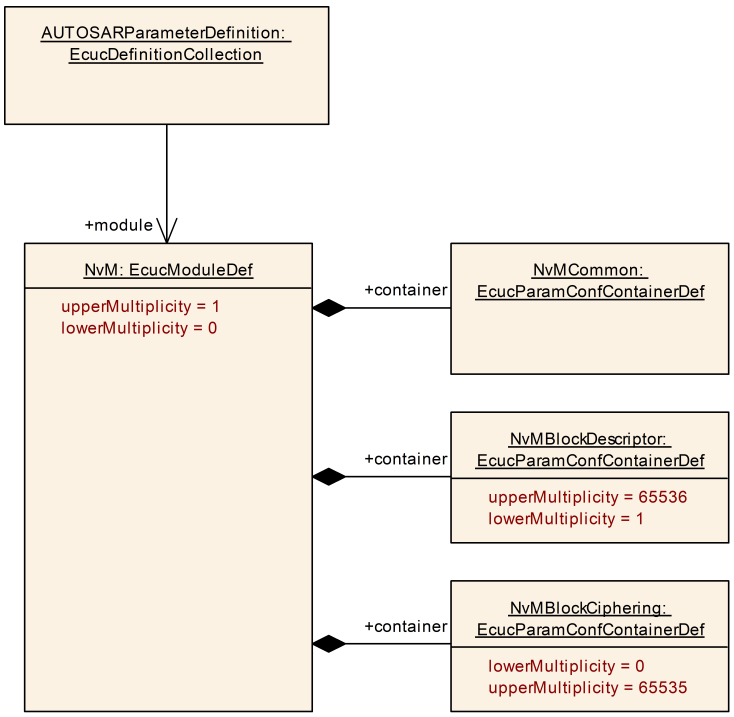

Figure 10.1: Containers and configuration parameters

10.2.1 NvM

Module SWS Item	ECUC_NvM_00539	
Module Name	NvM	
Module Description	Configuration of the NvM (NvRam Manager) module.	
Post-Build Variant Support	false	
Supported Config Variants	VARIANT-LINK-TIME, VARIANT-PRE-COMPILE	
Included Containers		
Container Name	**Multiplicity**	**Scope / Dependency**
NvMBlockCiphering	0..65535	Container for a chiphering of the Block. **Tags:** atp.Status=draft
NvMBlockDescriptor	1..65536	Container for a management structure to configure the composition of a given NVRAM Block Management Type. Its multiplicity describes the number of configured NVRAM blocks, one block is required to be configured. The NVRAM block descriptors are condensed in the NVRAM block descriptor table.
NvMCommon	1	Container for common configuration options.

Container Name	Multiplicity	Scope / Dependency
NvmDemEventParameterRefs	0..1	Container for the references to DemEventParameter elements which shall be invoked using the API Dem_SetEventStatus in case the corresponding error occurs. The EventId is taken from the referenced DemEventParameter's DemEventId symbolic value. The standardized errors are provided in this container and can be extended by vendor-specific error references.

10.2.2 NvMCommon

SWS Item	[ECUC_NvM_00028]
Container Name	NvMCommon
Parent Container	NvM
Description	Container for common configuration options.
Configuration Parameters	

Name	NvMApiConfigClass [ECUC_NvM_00491]		
Parent Container	NvMCommon		
Description	Preprocessor switch to enable some API calls which are related to NVM API configuration classes.		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	NVM_API_CONFIG_CLASS_1	All API calls belonging to configuration class 1 are available.	
	NVM_API_CONFIG_CLASS_2	All API calls belonging to configuration class 2 are available.	
	NVM_API_CONFIG_CLASS_3	All API calls belonging to configuration class 3 are available.	
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMBswMMultiBlockJobStatusInformation [ECUC_NvM_00550]
Parent Container	NvMCommon
Description	This parameter specifies whether BswM is informed about the current status of the multiblock job. True: call BswM_NvM_CurrentJobMode if ReadAll and WriteAll are started, finished, canceled False: do not inform BswM at all
Multiplicity	1
Type	EcucBooleanParamDef
Default Value	true
Post-Build Variant Value	false

Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMCompiledConfigId [ECUC_NvM_00492]		
Parent Container	NvMCommon		
Description	Configuration ID regarding the NV memory layout. This configuration ID shall be published as e.g. a SW-C shall have the possibility to write it to NV memory.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 65535		
Default Value			
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMCrcNumOfBytes [ECUC_NvM_00493]		
Parent Container	NvMCommon		
Description	If CRC is configured for at least one NVRAM block, this parameter defines the maximum number of bytes which shall be processed within one cycle of job processing.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	1 .. 65535		
Default Value			
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMCsmRetryCounter [ECUC_NvM_00572]		
Parent Container	NvMCommon		
Description	This value specifies the number of CSM encryption/decryption job retry attempts. CSM jobs for block reading and writing may fail (e.g. module busy, queue full, ...). To not directly abort the read/write with an error status, the NvM will retry the CSM job for the configured NvMCsmRetryCounter times. Configuring 0 means: no retry behavior; job will be aborted directly. **Tags:** atp.Status=draft		
Multiplicity	0..1		
Type	EcucIntegerParamDef		
Range	0 .. 255		
Default Value	0		
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMDatasetSelectionBits [ECUC_NvM_00494]		
Parent Container	NvMCommon		
Description	Defines the number of least significant bits which shall be used to address a certain dataset of a NVRAM block within the interface to the memory hardware abstraction. 0..8: Number of bits which are used for dataset or redundant block addressing. 0: No dataset or redundant NVRAM blocks are configured at all, no selection bits required. 1: In case of redundant NVRAM blocks are configured, but no dataset NVRAM blocks.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 8		
Default Value			
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local dependency: MemHwA, NVM_NVRAM_BLOCK_IDENTIFIER, NVM_BLOCK_MANAGEMENT_TYPE		

Name	NvMDevErrorDetect [ECUC_NvM_00495]		
Parent Container	NvMCommon		
Description	Switches the development error detection and notification on or off. • true: detection and notification is enabled. • false: detection and notification is disabled.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMDynamicConfiguration [ECUC_NvM_00497]		
Parent Container	NvMCommon		
Description	Preprocessor switch to enable the dynamic configuration management handling by the NvM_ReadAll request. true: Dynamic configuration management handling enabled. false: Dynamic configuration management handling disabled. This parameter affects all NvM processing related to Block with ID 1 and all processing related to Resistant to Changed Software. If the Dynamic Configuration is disabled, Block 1 cannot be used by NvM.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value			
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMJobPrioritization [ECUC_NvM_00498]		
Parent Container	NvMCommon		
Description	Preprocessor switch to enable job prioritization handling true: Job prioritization handling enabled. false: Job prioritization handling disabled.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value			
Post-Build Variant Value	false		

Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMMainFunctionPeriod [ECUC_NvM_00555]		
Parent Container	NvMCommon		
Description	The period between successive calls to the main function in seconds.		
Multiplicity	1		
Type	EcucFloatParamDef		
Range	]0 .. INF[		
Default Value			
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	NvMMultiBlockCallback [ECUC_NvM_00500]		
Parent Container	NvMCommon		
Description	Entry address of the common callback routine which shall be invoked on termination of each asynchronous multi block request		
Multiplicity	0..1		
Type	EcucFunctionNameDef		
Default Value			
Regular Expression			
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMPollingMode [ECUC_NvM_00501]		
Parent Container	NvMCommon		
Description	Preprocessor switch to enable/disable the polling mode in the NVRAM Manager and at the same time disable/enable the callback functions useable by lower layers true: Polling mode enabled, callback function usage disabled. false: Polling mode disabled, callback function usage enabled.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value			
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMRepeatMirrorOperations [ECUC_NvM_00518]		
Parent Container	NvMCommon		
Description	Defines the number of retries to let the application copy data to or from the NvM module's mirror before postponing the current job.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 7		
Default Value	0		
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMSetRamBlockStatusApi [ECUC_NvM_00502]		
Parent Container	NvMCommon		
Description	Preprocessor switch to enable the API NvM_SetRamBlockStatus. true: API NvM_SetRamBlockStatus enabled. false: API NvM_SetRamBlockStatus disabled.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value			
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	

Scope / Dependency	scope: local

Name	NvMSizeImmediateJobQueue [ECUC_NvM_00503]		
Parent Container	NvMCommon		
Description	Defines the number of queue entries for the immediate priority job queue. If NVM_JOB_PRIORITIZATION is switched OFF this parameter shall be out of scope.		
Multiplicity	0..1		
Type	EcucIntegerParamDef		
Range	1 .. 65535		
Default Value			
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local dependency: NVM_JOB_PRIORITIZATION		

Name	NvMSizeStandardJobQueue [ECUC_NvM_00504]		
Parent Container	NvMCommon		
Description	Defines the number of queue entries for the standard job queue.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	1 .. 65535		
Default Value			
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMVersionInfoApi [ECUC_NvM_00505]		
Parent Container	NvMCommon		
Description	Pre-processor switch to enable / disable the API to read out the modules version information]. true: Version info API enabled. false: Version info API disabled.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMBufferAlignmentValue [ECUC_NvM_00573]		
Parent Container	NvMCommon		
Description	Parameter determines the alignment of the start address that NvM buffers need to have. Value shall be inherited from EaBufferAlignmentValue or FeeBufferAlignmentValue. **Tags:** atp.Status=draft		
Multiplicity	1		
Type	Choice reference to [EaGeneral, FeeGeneral]		
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	NvMEcucPartitionRef [ECUC_NvM_00565]		
Parent Container	NvMCommon		
Description	Maps the NvM to one or multiple ECUC partitions to make its C-APIs available in the according partition.		
Multiplicity	1..*		
Type	Reference to EcucPartition		
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	NvMMasterEcucPartitionRef [ECUC_NvM_00566]
Parent Container	NvMCommon

Description	Maps the NvM master to zero or one ECUC partition to assign the master functionality to a certain core. The ECUC partition referenced is a subset of the ECUC partitions where the NvM is mapped to.		
Multiplicity	0..1		
Type	Reference to EcucPartition		
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

No Included Containers

[SWS_NvM_CONSTR_00974] ⌈The ECUC partition referenced by NvMMasterEcucPartitionRef shall be within the subset of the ECUC partitions referenced by NvMEcucPartitionRef.⌋*()*

[SWS_NvM_CONSTR_00975] ⌈If NvMEcucPartitionRef references one or more ECUC partitions, NvMMasterEcucPartitionRef shall have a multiplicity of one and reference one of these ECUC partitions as well.⌋*()*

10.2.3 NvMBlockDescriptor

SWS Item	[ECUC_NvM_00061]
Container Name	NvMBlockDescriptor
Parent Container	NvM
Description	Container for a management structure to configure the composition of a given NVRAM Block Management Type. Its multiplicity describes the number of configured NVRAM blocks, one block is required to be configured. The NVRAM block descriptors are condensed in the NVRAM block descriptor table.
Configuration Parameters	

Name	NvMBlockCrcType [ECUC_NvM_00476]	
Parent Container	NvMBlockDescriptor	
Description	Defines CRC data width for the NVRAM block. Default: NVM_CRC16, i.e. CRC16 will be used if NVM_BLOCK_USE_CRC==true	
Multiplicity	0..1	
Type	EcucEnumerationParamDef	
Range	NVM_CRC16	(Default) CRC16 will be used if NVM_BLOCK_USE_CRC==true.
	NVM_CRC32	CRC32 is selected for this NVRAM block if NVM_BLOCK_USE_CRC==true.
	NVM_CRC8	CRC8 is selected for this NVRAM block if NVM_BLOCK_USE_CRC==true.
Post-Build Variant Multiplicity	false	

Post-Build Variant Value	false		
Multiplicity Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local dependency: NVM_BLOCK_USE_CRC, NVM_CALC_RAM_BLOCK_CRC		

Name	NvMBlockHeaderInclude [ECUC_NvM_00554]		
Parent Container	NvMBlockDescriptor		
Description	Defines the header file where the owner of the NVRAM block has the declarations of the permanent RAM data block, ROM data block (if configured) and the callback function prototype for each configured callback. If no permanent RAM block, ROM block or callback functions are configured then this configuration parameter shall be ignored.		
Multiplicity	0..1		
Type	EcucStringParamDef		
Default Value			
Regular Expression			
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMBlockJobPriority [ECUC_NvM_00477]	
Parent Container	NvMBlockDescriptor	
Description	Defines the job priority for a NVRAM block (0 = Immediate priority).	
Multiplicity	1	
Type	EcucIntegerParamDef	
Range	0 .. 255	
Default Value		
Post-Build Variant Value	false	

Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMBlockManagementType [ECUC_NvM_00062]		
Parent Container	NvMBlockDescriptor		
Description	Defines the block management type for the NVRAM block.[SWS_NvM_00137]		
Multiplicity	1		
Type	EcucEnumerationParamDef		
Range	NVM_BLOCK_DATASET	NVRAM block is configured to be of dataset type.	
	NVM_BLOCK_NATIVE	NVRAM block is configured to be of native type.	
	NVM_BLOCK_REDUNDANT	NVRAM block is configured to be of redundant type.	
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMBlockUseAutoValidation [ECUC_NvM_00557]		
Parent Container	NvMBlockDescriptor		
Description	Defines whether the RAM Block shall be auto validated during shutdown phase. true: if auto validation mechanism is used, false: otherwise		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMBlockUseCompression [ECUC_NvM_00563]		
Parent Container	NvMBlockDescriptor		
Description	Defines whether the data is compressed before written. true: data compression activated (takes more time to read and write) false: no compression **Tags:** atp.Status=draft		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMBlockUseCrc [ECUC_NvM_00036]		
Parent Container	NvMBlockDescriptor		
Description	Defines CRC usage for the NVRAM block, i.e. memory space for CRC is reserved in RAM and NV memory. true: CRC will be used for this NVRAM block. false: CRC will not be used for this NVRAM block.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value			
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMBlockUseCRCCompMechanism [ECUC_NvM_00556]
Parent Container	NvMBlockDescriptor
Description	Defines whether the CRC of the RAM Block shall be compared during a write job with the CRC which was calculated during the last successful read or write job. true: if compare mechanism is used, false: otherwise
Multiplicity	1
Type	EcucBooleanParamDef
Default Value	false
Post-Build Variant Value	false

Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local dependency: False if NvMBlockUseCrc = False		

Name	NvMBlockUsePort [ECUC_NvM_00559]		
Parent Container	NvMBlockDescriptor		
Description	If this parameter is true it defines whether: • the port with interface 'NvMMirror' for synchronization mechanism callbacks are generated if the parameter NvMBlockUseSyncMechanism is configured TRUE; • the port with interface 'NvMNotifyInitBlock' for initialization block callback is generated if NvMInitBlockCallback parameter is configured (independent of the content); • the port with interface 'NvMNotifyJobFinished' for single block callback is generated if NvMSingleBlockCallback parameter is configured (independent of the content); • the port with interface 'NvMAdmin' for SetBlockProtection operation is generated.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value			
Multiplicity Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMBlockUseSetRamBlockStatus [ECUC_NvM_00552]		
Parent Container	NvMBlockDescriptor		
Description	Defines if NvMSetRamBlockStatusApi shall be used for this block or not. Note: If NvMSetRamBlockStatusApi is disabled this configuration parameter shall be ignored. true: calling of NvMSetRamBlockStatus for this RAM block shall set the status of the RAM block. false: calling of NvMSetRamBlockStatus for this RAM block shall be ignored.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value			
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMBlockUseSyncMechanism [ECUC_NvM_00519]		
Parent Container	NvMBlockDescriptor		
Description	Defines whether an explicit synchronization mechanism with a RAM mirror and callback routines for transferring data to and from NvM module's RAM mirror is used for NV block. true if synchronization mechanism is used, false otherwise.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMBlockWriteProt [ECUC_NvM_00033]
Parent Container	NvMBlockDescriptor
Description	Defines an initial write protection of the NV block true: Initial block write protection is enabled. false: Initial block write protection is disabled.
Multiplicity	1
Type	EcucBooleanParamDef
Default Value	

Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMBswMBlockStatusInformation [ECUC_NvM_00551]		
Parent Container	NvMBlockDescriptor		
Description	This parameter specifies whether BswM is informed about the current status of the specified block. True: Call BswM_NvM_CurrentBlockMode on changes False: Don't inform BswM at all		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMCalcRamBlockCrc [ECUC_NvM_00119]		
Parent Container	NvMBlockDescriptor		
Description	Defines CRC (re)calculation for the permanent RAM block or NVRAM blocks which are configured to use explicit synchronization mechanism. true: CRC will be (re)calculated for this permanent RAM block. false: CRC will not be (re)calculated for this permanent RAM block.		
Multiplicity	0..1		
Type	EcucBooleanParamDef		
Default Value			
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local dependency: NVM_BLOCK_USE_CRC		

Name	NvMMaxNumOfReadRetries [ECUC_NvM_00533]		
Parent Container	NvMBlockDescriptor		
Description	Defines the maximum number of read retries.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 7		
Default Value	0		
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMMaxNumOfWriteRetries [ECUC_NvM_00499]		
Parent Container	NvMBlockDescriptor		
Description	Defines the maximum number of write retries for a NVRAM block with [ECUC_NvM_00061]. Regardless of configuration a consistency check (and maybe write retries) are always forced for each block which is processed by the request NvM_WriteAll and NvM_WriteBlock.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 7		
Default Value			
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMNvBlockBaseNumber [ECUC_NvM_00478]		
Parent Container	NvMBlockDescriptor		
Description	Configuration parameter to perform the link between the NVM_NVRAM_BLOCK_IDENTIFIER used by the SW-Cs and the FEE_BLOCK_NUMBER expected by the memory abstraction modules. The parameter value equals the FEE_BLOCK_NUMBER or EA_BLOCK_NUMBER shifted to the right by NvMDatasetSelectionBits bits. (ref. to chapter 7.1.2.1). Calculation Formula: value = TargetBlockReference.[Ea/Fee]BlockConfiguration.[Ea/Fee]BlockNumber » NvMDatasetSelectionBits		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	1 .. 65534		
Default Value			

Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local dependency: FEE_BLOCK_NUMBER, EA_BLOCK_NUMBER		

Name	NvMNvBlockLength [ECUC_NvM_00479]		
Parent Container	NvMBlockDescriptor		
Description	Defines the NV block data length in bytes. Note: The implementer can add the attribute 'withAuto' to the parameter definition which indicates that the length can be calculated by the generator automatically (e.g. by using a parser that searches and analyzes the data structure corresponding to the block). When 'withAuto' is set to 'true' for this parameter definition the 'isAutoValue' can be set to 'true'. If 'isAutoValue' is set to 'true' the actual value will not be considered during ECU Configuration but will be (re-)calculated by the code generator and stored in the value attribute afterwards.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	1 .. 65535		
Default Value			
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMNvBlockNum [ECUC_NvM_00480]		
Parent Container	NvMBlockDescriptor		
Description	Defines the number of multiple NV blocks in a contiguous area according to the given block management type. 1-255 For NVRAM blocks to be configured of block management type NVM_BLOCK_DATASET. The actual range is limited according to SWS_NvM_00444. 1 For NVRAM blocks to be configured of block management type NVM_BLOCK_NATIVE 2 For NVRAM blocks to be configured of block management type NVM_BLOCK_REDUNDANT		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	1 .. 255		

Default Value			
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local dependency: NVM_BLOCK_MANAGEMENT_TYPE		

Name	NvMNvramBlockIdentifier [ECUC_NvM_00481]		
Parent Container	NvMBlockDescriptor		
Description	Identification of a NVRAM block via a unique block identifier. Implementation Type: NvM_BlockIdType. min = 2 max = 2^ (16- NVM_DATASET_SELECTION_BITS)-1 Reserved NVRAM block IDs: 0 -> to derive multi block request results via NvM_GetErrorStatus 1 -> redundant NVRAM block which holds the configuration ID (generation tool should check that this block is correctly configured from type,CRC and size point of view)		
Multiplicity	1		
Type	EcucIntegerParamDef (Symbolic Name generated for this parameter)		
Range	2 .. 65535		
Default Value			
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: local dependency: NVM_DATASET_SELECTION_BITS		

Name	NvMNvramDeviceId [ECUC_NvM_00035]		
Parent Container	NvMBlockDescriptor		
Description	Defines the NVRAM device ID where the NVRAM block is located. Calculation Formula: value = TargetBlockReference.[Ea/Fee]DeviceIndex		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 1		
Default Value			
Post-Build Variant Value	false		

Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local dependency: EA_DEVICE_INDEX, FEE_DEVICE_INDEX		

Name	NvMRamBlockDataAddress [ECUC_NvM_00482]		
Parent Container	NvMBlockDescriptor		
Description	Defines the start address of the RAM block data. If this is not configured, no permanent RAM data block is available for the selected block management type.		
Multiplicity	0..1		
Type	EcucStringParamDef		
Default Value			
Regular Expression			
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMReadRamBlockFromNvCallback [ECUC_NvM_00521]
Parent Container	NvMBlockDescriptor
Description	Entry address of a block specific callback routine which shall be called in order to let the application copy data from the NvM module's mirror to RAM block. Implementation type: Std_ReturnType E_OK: copy was successful E_NOT_OK: copy was not successful, callback routine to be called again
Multiplicity	0..1
Type	EcucFunctionNameDef
Default Value	
Regular Expression	
Post-Build Variant Multiplicity	false
Post-Build Variant Value	false

Multiplicity Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMResistantToChangedSw [ECUC_NvM_00483]		
Parent Container	NvMBlockDescriptor		
Description	Defines whether a NVRAM block shall be treated resistant to configuration changes or not. If there is no default data available at configuration time then the application shall be responsible for providing the default initialization data. In this case the application has to use NvM_GetErrorStatus()to be able to distinguish between first initialization and corrupted data. true: NVRAM block is resistant to changed software. false: NVRAM block is not resistant to changed software.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value			
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMRomBlockDataAddress [ECUC_NvM_00484]		
Parent Container	NvMBlockDescriptor		
Description	Defines the start address of the ROM block data. If not configured, no ROM block is available for the selected block management type.		
Multiplicity	0..1		
Type	EcucStringParamDef		
Default Value			
Regular Expression			
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	

Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMRomBlockNum [ECUC_NvM_00485]		
Parent Container	NvMBlockDescriptor		
Description	Defines the number of multiple ROM blocks in a contiguous area according to the given block management type. 0-254 For NVRAM blocks to be configured of block management type NVM_BLOCK_DATASET. The actual range is limited according to SWS_NvM_00444. 0-1 For NVRAM blocks to be configured of block management type NVM_BLOCK_NATIVE 0-1 For NVRAM blocks to be configured of block management type NVM_BLOCK_REDUNDANT		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 254		
Default Value			
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local dependency: NVM_BLOCK_MANAGEMENT_TYPE, NVM_NV_BLOCK_NUM		

Name	NvMSelectBlockForFirstInitAll {NVM_SELECT_BLOCK_FOR_FIRST_INIT_ALL} [ECUC_NvM_00558]
Parent Container	NvMBlockDescriptor
Description	Defines whether a block will be processed or not by NvM_FirstInitAll. A block can be configured to be processed even if it doesn't have permanent RAM and/or explicit synchronization. TRUE: block will be processed by NvM_FirstInitAll FALSE: block will not be processed by NvM_FirstInitAll
Multiplicity	0..1
Type	EcucBooleanParamDef
Default Value	false
Post-Build Variant Multiplicity	false
Post-Build Variant Value	false

Multiplicity Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMSelectBlockForReadAll [ECUC_NvM_00117]		
Parent Container	NvMBlockDescriptor		
Description	Defines whether a NVRAM block shall be processed during NvM_ReadAll or not. This configuration parameter has only influence on those NVRAM blocks which are configured to have a permanent RAM block or which are configured to use explicit synchronization mechanism. true: NVRAM block shall be processed by NvM_ReadAll false: NVRAM block shall not be processed by NvM_ReadAll		
Multiplicity	0..1		
Type	EcucBooleanParamDef		
Default Value			
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local dependency: NVM_RAM_BLOCK_DATA_ADDRESS		

Name	NvMSelectBlockForWriteAll [ECUC_NvM_00549]		
Parent Container	NvMBlockDescriptor		
Description	Defines whether a NVRAM block shall be processed during NvM_WriteAll or not. This configuration parameter has only influence on those NVRAM blocks which are configured to have a permanent RAM block or which are configured to use explicit synchronization mechanism. true: NVRAM block shall be processed by NvM_WriteAll false: NVRAM block shall not be processed by NvM_WriteAll		
Multiplicity	0..1		
Type	EcucBooleanParamDef		
Default Value			
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local dependency: NVM_RAM_BLOCK_DATA_ADDRESS		

Name	NvMStaticBlockIDCheck [ECUC_NvM_00532]		
Parent Container	NvMBlockDescriptor		
Description	Defines if the Static Block ID check is enabled. false: Static Block ID check is disabled. true: Static Block ID check is enabled.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMWriteBlockOnce [ECUC_NvM_00072]		
Parent Container	NvMBlockDescriptor		
Description	Defines write protection after first write. The NVRAM manager sets the write protection bit either after the NV block was written the first time or if the block was already written and it is detected as valid and consistent during a read for it. true: Defines write protection after first write is enabled. false: Defines write protection after first write is disabled.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value			
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMWriteRamBlockToNvCallback [ECUC_NvM_00520]		
Parent Container	NvMBlockDescriptor		
Description	Entry address of a block specific callback routine which shall be called in order to let the application copy data from RAM block to NvM module's mirror. Implementation type: Std_ReturnType E_OK: copy was successful E_NOT_OK: copy was not successful, callback routine to be called again		
Multiplicity	0..1		
Type	EcucFunctionNameDef		
Default Value			
Regular Expression			
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMWriteVerification [ECUC_NvM_00534]		
Parent Container	NvMBlockDescriptor		
Description	Defines if Write Verification is enabled. false: Write verification is disabled. true: Write Verification is enabled.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default Value	false		
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMWriteVerificationDataSize [ECUC_NvM_00538]		
Parent Container	NvMBlockDescriptor		
Description	Defines the number of bytes to compare in each step when comparing the content of a RAM Block and a block read back.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	1 .. 65535		
Default Value			
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMBlockCipheringRef [ECUC_NvM_00567]
Parent Container	NvMBlockDescriptor
Description	Reference to ciphering container. If configured, NvM encrypt the data before storage and decrypt the data after restoring. If empty, the NvM stores and restore the original user data. **Tags:** atp.Status=draft
Multiplicity	0..1
Type	Reference to NvMBlockCiphering
Post-Build Variant Multiplicity	false
Post-Build Variant Value	false

Multiplicity Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: local dependency: Key will be located in RAM if this configuration item is not present.		

Name	NvMBlockEcucPartitionRef [ECUC_NvM_00564]		
Parent Container	NvMBlockDescriptor		
Description	Maps the NV block to zero or one ECUC partition to limit the access to this NV block. The ECUC partition referenced is within the subset of the ECUC partitions where the NvM is mapped to.		
Multiplicity	0..1		
Type	Reference to EcucPartition		
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Included Containers		
Container Name	**Multiplicity**	**Scope / Dependency**
NvMInitBlockCallback	0..1	The presence of this container indicates, that a block specific callback routine is called if no ROM data is available for initialization of the NVRAM block. If the container is not present, no callback routine is called for initialization of the NVRAM block with default data. In case the container has a NvMInitBlockCallbackFnc, the NvM will call this function. In case there is no NvMInitBlockCallbackFnc, the NvM will have an port PNIB_{Block}.
NvMSingleBlockCallback	0..1	The presence of this container indicates, that the block specific callback routine which shall be invoked on termination of each asynchronous single block request [SWS_NvM_00113] If the container is not present, no callback routine is called.. In case the container has a NvMSingleBlockCallbackFnc, the NvM will call this function. In case there is no NvMSingleBlockCallbackFnc, the NvM will have an port PNJF_{Block}.

NvMTargetBlock Reference	1	This parameter is just a container for the parameters for EA and FEE

[SWS_NvM_CONSTR_00972] ⌈The ECUC partition referenced by NvMBlockEcuc PartitionRef shall be within the subset of the ECUC partitions referenced by NvMEcuc PartitionRef.⌋*()*

[SWS_NvM_CONSTR_00973] ⌈If NvMEcucPartitionRef references two or more ECUC partitions, NvMBlockEcucPartitionRef shall have a multiplicity one and reference one of these ECUC partitions as well.⌋*()*

10.2.4 NvMInitBlockCallback

SWS Item	[ECUC_NvM_00561]		
Container Name	NvMInitBlockCallback		
Parent Container	NvMBlockDescriptor		
Description	The presence of this container indicates, that a block specific callback routine is called if no ROM data is available for initialization of the NVRAM block. If the container is not present, no callback routine is called for initialization of the NVRAM block with default data. In case the container has a NvMInitBlockCallbackFnc, the NvM will call this function. In case there is no NvMInitBlockCallbackFnc, the NvM will have an port PNIB_{Block}.		
Multiplicity Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Configuration Parameters			

Name	NvMInitBlockCallbackFnc [ECUC_NvM_00116]
Parent Container	NvMInitBlockCallback
Description	Entry address of a block specific callback routine which shall be called if no ROM data is available for initialization of the NVRAM block. If not configured, no specific callback routine shall be called for initialization of the NVRAM block with default data.
Multiplicity	0..1
Type	EcucFunctionNameDef
Default Value	
Regular Expression	
Post-Build Variant Multiplicity	false
Post-Build Variant Value	false

Multiplicity Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

No Included Containers

10.2.5 NvMSingleBlockCallback

SWS Item	[ECUC_NvM_00562]		
Container Name	NvMSingleBlockCallback		
Parent Container	NvMBlockDescriptor		
Description	The presence of this container indicates, that the block specific callback routine which shall be invoked on termination of each asynchronous single block request [SWS_NvM_00113] If the container is not present, no callback routine is called.. In case the container has a NvMSingleBlockCallbackFnc, the NvM will call this function. In case there is no NvMSingleBlockCallbackFnc, the NvM will have an port PNJF_{Block}.		
Multiplicity Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Configuration Parameters			

Name	NvMSingleBlockCallbackFnc [ECUC_NvM_00506]
Parent Container	NvMSingleBlockCallback
Description	Entry address of the block specific callback routine which shall be invoked on termination of each asynchronous single block request [SWS_NvM_00113].
Multiplicity	0..1
Type	EcucFunctionNameDef
Default Value	
Regular Expression	
Post-Build Variant Multiplicity	false
Post-Build Variant Value	false

Multiplicity Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

No Included Containers

10.2.6 NvMTargetBlockReference

SWS Item	[ECUC_NvM_00486]
Container Name	NvMTargetBlockReference
Parent Container	NvMBlockDescriptor
Description	This parameter is just a container for the parameters for EA and FEE
Configuration Parameters	

Container Choices		
Container Name	**Multiplicity**	**Scope / Dependency**
NvMEaRef	0..1	EEPROM Abstraction
NvMFeeRef	0..1	Flash EEPROM Emulation

10.2.7 NvMEaRef

SWS Item	[ECUC_NvM_00487]
Container Name	NvMEaRef
Parent Container	NvMTargetBlockReference
Description	EEPROM Abstraction
Configuration Parameters	

Name	NvMNameOfEaBlock [ECUC_NvM_00488]		
Parent Container	NvMEaRef		
Description	reference to EaBlock		
Multiplicity	1		
Type	Symbolic name reference to EaBlockConfiguration		
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

No Included Containers

10.2.8 NvMFeeRef

SWS Item	[ECUC_NvM_00489]		
Container Name	NvMFeeRef		
Parent Container	NvMTargetBlockReference		
Description	Flash EEPROM Emulation		
Configuration Parameters			

Name	NvMNameOfFeeBlock [ECUC_NvM_00490]		
Parent Container	NvMFeeRef		
Description	reference to FeeBlock		
Multiplicity	1		
Type	Symbolic name reference to FeeBlockConfiguration		
Post-Build Variant Value	false		
Value Configuration Class	Pre-compile time	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	–	
Scope / Dependency	scope: local		

No Included Containers

10.2.9 NvmDemEventParameterRefs

SWS Item	[ECUC_NvM_00541]
Container Name	NvmDemEventParameterRefs
Parent Container	NvM
Description	Container for the references to DemEventParameter elements which shall be invoked using the API Dem_SetEventStatus in case the corresponding error occurs. The EventId is taken from the referenced DemEventParameter's DemEventId symbolic value. The standardized errors are provided in this container and can be extended by vendor-specific error references.
Configuration Parameters	

Name	NVM_E_HARDWARE [ECUC_NvM_00553]		
Parent Container	NvmDemEventParameterRefs		
Description	Reference to the DemEventParameter which shall be issued when the hardware error has occured.		
Multiplicity	0..1		
Type	Symbolic name reference to DemEventParameter		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	NVM_E_INTEGRITY_FAILED [ECUC_NvM_00542]		
Parent Container	NvmDemEventParameterRefs		
Description	Reference to the DemEventParameter which shall be issued when the error "API request integrity failed" has occurred.		
Multiplicity	0..1		
Type	Symbolic name reference to DemEventParameter		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	NVM_E_LOSS_OF_REDUNDANCY [ECUC_NvM_00546]		
Parent Container	NvmDemEventParameterRefs		
Description	Reference to the DemEventParameter which shall be issued when the error "loss of redundancy" has occurred.		
Multiplicity	0..1		
Type	Symbolic name reference to DemEventParameter		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		

Multiplicity Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	NVM_E_REQ_FAILED [ECUC_NvM_00543]		
Parent Container	NvmDemEventParameterRefs		
Description	Reference to the DemEventParameter which shall be issued when the error "API request failed" has occurred.		
Multiplicity	0..1		
Type	Symbolic name reference to DemEventParameter		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	NVM_E_VERIFY_FAILED [ECUC_NvM_00545]		
Parent Container	NvmDemEventParameterRefs		
Description	Reference to the DemEventParameter which shall be issued when the error "Write Verification failed" has occurred.		
Multiplicity	0..1		
Type	Symbolic name reference to DemEventParameter		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	NVM_E_WRITE_PROTECTED [ECUC_NvM_00548]		
Parent Container	NvmDemEventParameterRefs		
Description	Reference to the DemEventParameter which shall be issued when the error "write attempt to NVRAM block with write protection" has occurred.		
Multiplicity	0..1		
Type	Symbolic name reference to DemEventParameter		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	NVM_E_WRONG_BLOCK_ID [ECUC_NvM_00544]		
Parent Container	NvmDemEventParameterRefs		
Description	Reference to the DemEventParameter which shall be issued when the error "Static Block ID check failed" has occurred.		
Multiplicity	0..1		
Type	Symbolic name reference to DemEventParameter		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

No Included Containers

10.2.10 NvMBlockCiphering

SWS Item	[ECUC_NvM_00568]
Container Name	NvMBlockCiphering
Parent Container	NvM

Description	Container for a chiphering of the Block. **Tags:** atp.Status=draft		
Post-Build Variant Multiplicity	false		
Multiplicity Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Configuration Parameters			

Name	NvMNvBlockNVRAMDataLength [ECUC_NvM_00569]		
Parent Container	NvMBlockCiphering		
Description	This value specified the block length in case ciphered data is stored in NV RAM - in that case the plain data length may not match the ciphered data length. This value needs be filled out, if the ciphering, encryption and decryption, of the data is enabled for this block. It will be used instead of the NvMNvBlockLength to access the NV RAM. Hint: This value can be equal or bigger than 'NvMNvBlockLength' depending on the characteristics of the referenced CSM Job inside 'NvMNameOfEncryptionJob' (e.g. CSM job adds padding information). **Tags:** atp.Status=draft		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	1 .. 65535		
Default Value	16		
Post-Build Variant Value	false		
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: local		

Name	NvMCsmDecryptionJobReference [ECUC_NvM_00571]		
Parent Container	NvMBlockCiphering		
Description	This parameter references a CSM decrypt job used to decrypt ciphered data after reading it from NV RAM. **Tags:** atp.Status=draft		
Multiplicity	1		
Type	Symbolic name reference to CsmJob		
Post-Build Variant Multiplicity	false		

Post-Build Variant Value	false		
Multiplicity Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

Name	NvMCsmEncryptionJobReference [ECUC_NvM_00570]		
Parent Container	NvMBlockCiphering		
Description	This parameter references a CSM encrypt job used to encrypt plain data before writing it to the NV RAM. **Tags:** atp.Status=draft		
Multiplicity	1		
Type	Symbolic name reference to CsmJob		
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: ECU		

No Included Containers

[SWS_NvM_00030] ⌈By use of configuration techniques, each application shall be enabled to declare the memory requirements at configuration time. This information shall be useable to assign memory areas and to generate the appropriate interfaces. Wrong memory assignments and conflicts in requirements (sufficient memory not available) shall be detected at configuration time.⌋*()*

[SWS_NvM_00034] ⌈The NVRAM memory layout configuration shall have a unique ID. The NvM module shall have a configuration identifier that is a unique property of the memory layout configuration. The ID can be either statically assigned to the configuration or it can be calculated from the configuration properties. This should be supported by a configuration tool. The ID should be changed if the block configuration changes, i.e. if a block is added or removed, or if its size or type is changed. The ID shall be

stored together with the data and shall be used in addition to the data checksum to determine the consistency of the NVRAM contents.⌋*(SRS_Mem_00135)*

[SWS_NvM_00073] ⌈The comparison between the stored configuration ID and the compiled configuration ID shall be done as the first step within the function NvM_Read All during startup.⌋*()*

[SWS_NvM_00688] ⌈In case of a detected configuration ID mismatch, the behavior of the NvM module shall be defined by a configurable option.⌋*()*

[SWS_NvM_00052] ⌈Provide information about used memory resources. The NvM module configuration shall provide information on how many resources of RAM, ROM and NVRAM are used. The configuration tool shall be responsible to provide detailed information about all reserved resources. The format of this information shall be commonly used (e.g. MAP file format).⌋*()*

10.3 Published Information

For details refer to the chapter 10.3 “Published Information” in SWS_BSWGeneral. [ref. to doc. [2]]

A Not applicable requirements

[SWS_NvM_00744] ⌈These requirements are not applicable to this specification.⌋*(SRS_BSW_00344, SRS_BSW_00404, SRS_BSW_00405, SRS_BSW_00170, SRS_BSW_00412, SRS_BSW_00398, SRS_BSW_00399, SRS_BSW_00400, SRS_BSW_00416, SRS_BSW_00168, SRS_BSW_00423, SRS_BSW_00426, SRS_BSW_00427, SRS_BSW_00432, SRS_BSW_00375, SRS_BSW_00422, SRS_BSW_00417, SRS_BSW_00336, SRS_BSW_00161, SRS_BSW_00162, SRS_BSW_00005, SRS_BSW_00415, SRS_BSW_00164, SRS_BSW_00325, SRS_BSW_00342, SRS_BSW_00343, SRS_BSW_00160, SRS_BSW_00007, SRS_BSW_00347, SRS_BSW_00307, SRS_BSW_00335, SRS_BSW_00314, SRS_BSW_00348, SRS_BSW_00353, SRS_BSW_00361, SRS_BSW_00302, SRS_BSW_00328, SRS_BSW_00312, SRS_BSW_00006, SRS_BSW_00304, SRS_BSW_00378, SRS_BSW_00306, SRS_BSW_00308, SRS_BSW_00309, SRS_BSW_00371, SRS_BSW_00330, SRS_BSW_00009, SRS_BSW_00010, SRS_BSW_00321, SRS_BSW_00341, SRS_BSW_00334)*