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

Document Version	3.2.0
Document Status	Final
Part of Release	4.0
Revision	3

Document Change History

Date	Version	Changed by	Change Description
02.11.2011	3.2.0	AUTOSAR Administration	• 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
25.10.2010	3.1.0	AUTOSAR Administration	• 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

30.11.2009	3.0.0	AUTOSAR Administration	• The following features had impact on this document: o Debugging concept o Error handler concept o Memory related concepts • The following major features were necessary to implement these concepts: o Static Block Id Check o Write Verification o Read Retry o buffered read/write-operations • Legal disclaimer revised
23.06.2008	2.1.1	AUTOSAR Administration	Legal disclaimer revised
26.01.2007	2.1.0	AUTOSAR Administration	• 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
28.04.2006	2.0.0	AUTOSAR Administration	• Document structure adapted to common Release 2.0 SWS Template. • Major changes in chapter 10 • Structure of document changed partly
20.06.2005	1.0.0	AUTOSAR Administration	Initial release

Disclaimer

Advice for users

AUTOSAR specifications may contain exemplary items (exemplary reference models, "use cases", and/or references to exemplary technical solutions, devices, processes or software).

Any such exemplary items are contained in the specifications for illustration purposes only, and they themselves are not part of the AUTOSAR Standard. Neither their presence in such specifications, nor any later documentation of AUTOSAR conformance of products actually implementing such exemplary items, imply that intellectual property rights covering such exemplary items are licensed under the same rules as applicable to the AUTOSAR Standard.

Table of Contents

Figures

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:

-

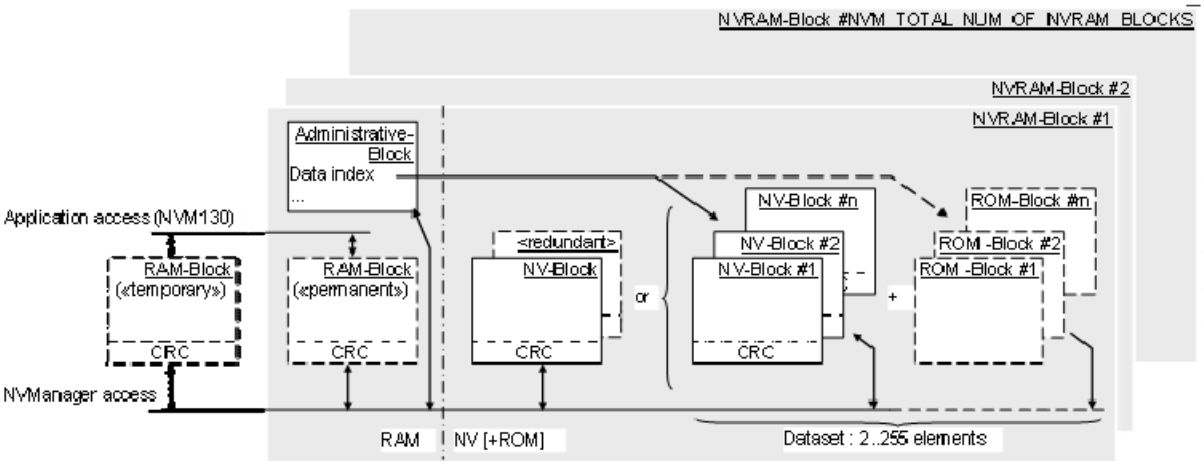

Figure 1: Memory Structure of Different Block Types

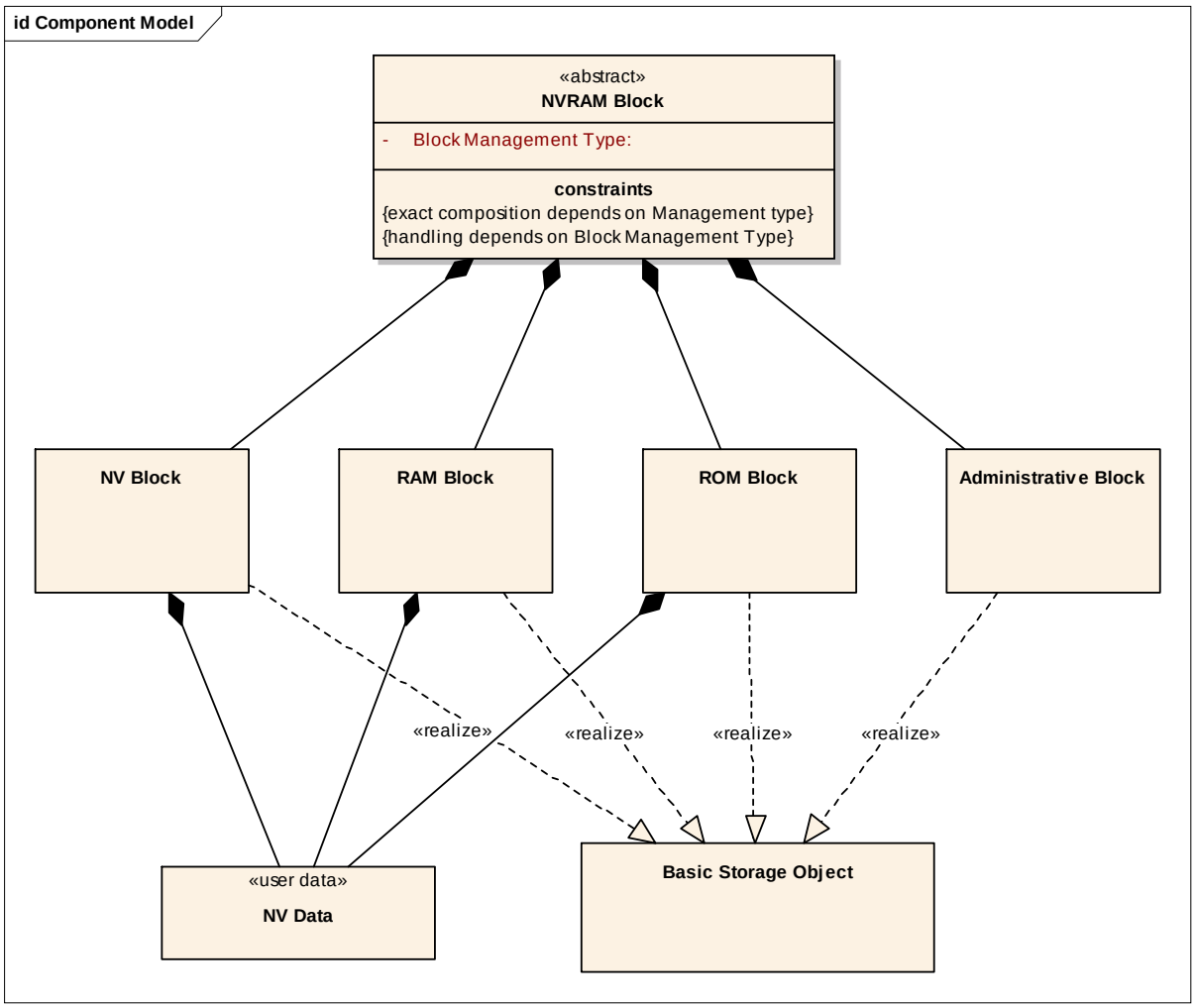

Figure 2: Logical Structure of Different Block Types

2 Acronyms and abbreviations

Acronyms and abbreviations, which have a local scope and therefore are not contained in the AUTOSAR glossary, must appear in a local 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 [BSW08534] .[NVM126]
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“.[NVM020]
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“. [NVM125]
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”. [NVM135]
DET	Development 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

[1] List of Basic Software Modules
AUTOSAR_TR_BSWModuleList.pdf

[2] Layered Software Architecture
AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf

[3] General Requirements on Basic Software Modules
AUTOSAR_SRS_BSWGeneral.pdf

[4] Requirements on Memory Services
AUTOSAR_SRS_MemoryServices.pdf

[5] Specification of EEPROM Abstraction
AUTOSAR_SWS_EEPROMAbstraction

[6] Specification of Flash EEPROM Emulation
AUTOSAR_SWS_FlashEEPROMEmulation

[7] Specification of Memory Abstraction Interface
AUTOSAR_SWS_MemoryAbstractionInterface

[8] Specification of Memory Mapping
AUTOSAR_SWS_MemoryMapping

[9] Virtual Functional Bus
AUTOSAR_EXP_VFB.pdf

[10] Software Component Template
AUTOSAR_TPS_SoftwareComponentTemplate

[11] Specification of RTE Software
AUTOSAR_SWS_RTE.pdf

[12] Specification of ECU Configuration
AUTOSAR_TPS_ECUConfiguration.pdf

[13] Basic Software Module Description Template
AUTOSAR_TPS_BSWModuleDescriptionTemplate

[14] Specification of CRC Routines
AUTOSAR_SWS_CRCLibrary

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

None

5 Dependencies to other modules

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

5.1 File structure

5.1.1 Code file structure

[NVM076] ⌈The NvM module shall consist of one or more C file NvM_xxx.c containing the entire or parts of NVRAM manager code⌋ ( )

5.1.2 Header file structure

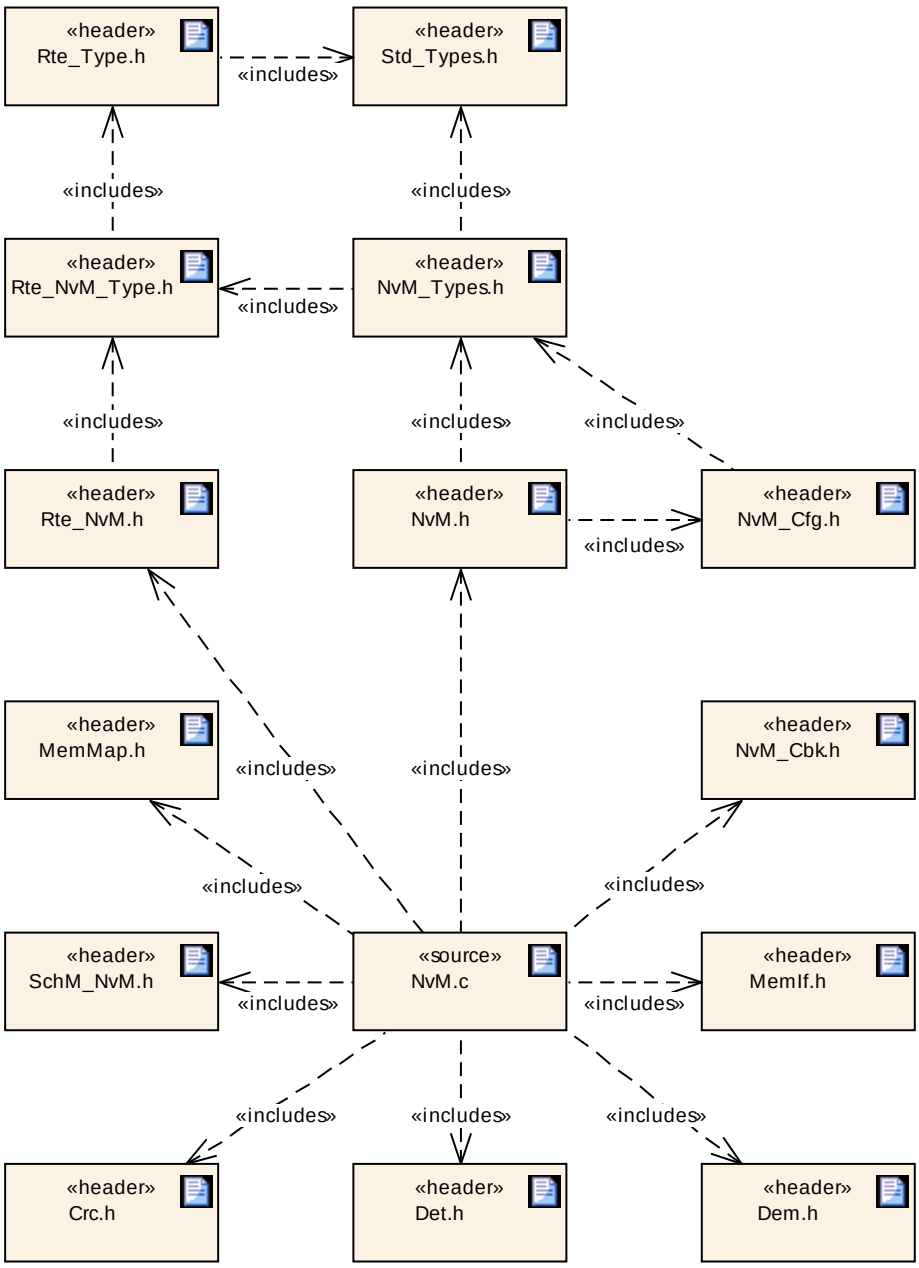

Figure 3: NvM Include structure

The include file structure shall be as follows:

[NVM077] ⌈An API interface NvM.h that provides the function prototypes to access the underlying NVRAM functions. ⌋(BSW00435, BSW00436)

[NVM550] ⌈A type header NvM_Types.h that provides the types for the NvM module. ⌋( )

[NVM755] ⌈The file NvM_Types.h shall include Rte_NvM_Type.h to include the types which are common used by BSW Modules and Software Components. NvM_Types.h and NvM.h shall only contain types, that are not already defined in Rte_NvM_Type.h. ⌋(BSW00447)

[NVM551] ⌈A callback interface NvM_Cbk.h that provides the callback function prototypes to be used by the lower layers ⌋( )

[NVM552] ⌈A type header NvM_Cfg.h that provides the configuration parameters for the NvM module. ⌋( )

[NVM689] ⌈NvM_Cfg.h shall include NvM_Types.h. ⌋( )

[NVM690] ⌈NvM_Types.h shall include Std_Types.h. ⌋( )

[NVM553] ⌈NvM.h shall include NvM_Cfg.h. ⌋( )

[NVM554] ⌈NvM module shall include NvM.h, Dem.h, MemIf.h, SchM_NvM.h, MemMap.h. ⌋( )

[NVM555] ⌈NvM module shall include Crc.h. ⌋( )

[NVM556] ⌈NvM module shall include Det.h. ⌋( )

[NVM691] ⌈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. 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.

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. 8.6.2].

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.

6 Requirements traceability

Requirement	Satisfied by
-	NVM380
-	NVM201
-	NVM443
-	NVM510
-	NVM628
-	NVM351
-	NVM561
-	NVM379
-	NVM245
-	NVM307
-	NVM192
-	NVM541
-	NVM377
-	NVM258
-	NVM737
-	NVM265
-	NVM375
-	NVM314
-	NVM717
-	NVM239
-	NVM731
-	NVM554
-	NVM704
-	NVM295
-	NVM661
-	NVM631
-	NVM647
-	NVM733
-	NVM369
-	NVM290
-	NVM680
-	NVM366
-	NVM279
-	NVM229
-	NVM603
-	NVM677
-	NVM697
-	NVM575

-	NVM091
-	NVM474
-	NVM598
-	NVM738
-	NVM306
-	NVM514
-	NVM434
-	NVM752
-	NVM726
-	NVM315
-	NVM370
-	NVM716
-	NVM391
-	NVM691
-	NVM447
-	NVM678
-	NVM635
-	NVM310
-	NVM014
-	NVM630
-	NVM364
-	NVM234
-	NVM572
-	NVM544
-	NVM333
-	NVM202
-	NVM121
-	NVM225
-	NVM513
-	NVM735
-	NVM255
-	NVM135
-	NVM724
-	NVM418
-	NVM643
-	NVM619
-	NVM607
-	NVM522
-	NVM689
-	NVM427
-	NVM642
-	NVM149

-	NVM189
-	NVM288
-	NVM574
-	NVM720
-	NVM356
-	NVM339
-	NVM301
-	NVM683
-	NVM625
-	NVM001
-	NVM608
-	NVM390
-	NVM546
-	NVM259
-	NVM127
-	NVM347
-	NVM113
-	NVM658
-	NVM556
-	NVM420
-	NVM374
-	NVM233
-	NVM452
-	NVM236
-	NVM272
-	NVM624
-	NVM209
-	NVM585
-	NVM637
-	NVM396
-	NVM287
-	NVM715
-	NVM111
-	NVM281
-	NVM134
-	NVM695
-	NVM143
-	NVM436
-	NVM349
-	NVM667
-	NVM614
-	NVM176

-	NVM671
-	NVM645
-	NVM725
-	NVM611
-	NVM385
-	NVM636
-	NVM685
-	NVM712
-	NVM217
-	NVM732
-	NVM274
-	NVM228
-	NVM580
-	NVM269
-	NVM706
-	NVM112
-	NVM431
-	NVM264
-	NVM076
-	NVM471
-	NVM146
-	NVM150
-	NVM198
-	NVM126
-	NVM739
-	NVM600
-	NVM302
-	NVM247
-	NVM605
-	NVM552
-	NVM469
-	NVM309
-	NVM248
-	NVM136
-	NVM530
-	NVM298
-	NVM682
-	NVM373
-	NVM346
-	NVM227
-	NVM710
-	NVM334

-	NVM292
-	NVM423
-	NVM206
-	NVM358
-	NVM260
-	NVM338
-	NVM311
-	NVM381
-	NVM703
-	NVM639
-	NVM417
-	NVM294
-	NVM232
-	NVM591
-	NVM368
-	NVM038
-	NVM612
-	NVM626
-	NVM602
-	NVM679
-	NVM184
-	NVM395
-	NVM730
-	NVM742
-	NVM389
-	NVM722
-	NVM622
-	NVM745
-	NVM470
-	NVM305
-	NVM383
-	NVM387
-	NVM550
-	NVM342
-	NVM702
-	NVM280
-	NVM648
-	NVM304
-	NVM237
-	NVM251
-	NVM313
-	NVM130

-	NVM604
-	NVM160
-	NVM363
-	NVM271
-	NVM616
-	NVM254
-	NVM020
-	NVM308
-	NVM468
-	NVM551
-	NVM256
-	NVM734
-	NVM140
-	NVM224
-	NVM376
-	NVM357
-	NVM747
-	NVM361
-	NVM262
-	NVM686
-	NVM328
-	NVM128
-	NVM595
-	NVM666
-	NVM275
-	NVM672
-	NVM525
-	NVM597
-	NVM360
-	NVM531
-	NVM654
-	NVM393
-	NVM187
-	NVM088
-	NVM587
-	NVM512
-	NVM175
-	NVM711
-	NVM569
-	NVM316
-	NVM718
-	NVM440

-	NVM665
-	NVM144
-	NVM632
-	NVM719
-	NVM249
-	NVM199
-	NVM701
-	NVM367
-	NVM085
-	NVM601
-	NVM204
-	NVM620
-	NVM180
-	NVM652
-	NVM664
-	NVM300
-	NVM238
-	NVM352
-	NVM592
-	NVM433
-	NVM650
-	NVM537
-	NVM372
-	NVM270
-	NVM212
-	NVM515
-	NVM567
-	NVM707
-	NVM651
-	NVM555
-	NVM235
-	NVM216
-	NVM663
-	NVM129
-	NVM644
-	NVM659
-	NVM586
-	NVM006
-	NVM432
-	NVM273
-	NVM445
-	NVM475

-	NVM230
-	NVM562
-	NVM185
-	NVM231
-	NVM610
-	NVM693
-	NVM736
-	NVM753
-	NVM430
-	NVM740
-	NVM422
-	NVM246
-	NVM226
-	NVM583
-	NVM721
-	NVM291
-	NVM547
-	NVM754
-	NVM618
-	NVM517
-	NVM257
-	NVM560
-	NVM655
-	NVM606
-	NVM509
-	NVM168
-	NVM657
-	NVM741
-	NVM511
-	NVM158
-	NVM653
-	NVM539
-	NVM708
-	NVM355
-	NVM092
-	NVM350
-	NVM392
-	NVM596
-	NVM329
-	NVM263
-	NVM579
-	NVM438

-	NVM444
-	NVM570
-	NVM746
-	NVM441
-	NVM312
-	NVM341
-	NVM615
-	NVM155
-	NVM200
-	NVM670
-	NVM681
-	NVM549
-	NVM599
-	NVM243
-	NVM542
-	NVM047
-	NVM139
-	NVM354
-	NVM156
-	NVM553
-	NVM388
-	NVM649
-	NVM673
-	NVM359
-	NVM750
-	NVM462
-	NVM408
-	NVM040
-	NVM118
-	NVM296
-	NVM138
-	NVM705
-	NVM684
-	NVM694
-	NVM590
-	NVM424
-	NVM426
-	NVM210
-	NVM318
-	NVM467
-	NVM133
-	NVM709

-	NVM054
-	NVM394
-	NVM573
-	NVM181
-	NVM723
-	NVM714
-	NVM293
-	NVM284
-	NVM578
-	NVM568
-	NVM125
-	NVM729
-	NVM449
-	NVM303
-	NVM674
-	NVM416
-	NVM748
-	NVM406
-	NVM646
-	NVM343
-	NVM728
-	NVM446
-	NVM435
-	NVM690
-	NVM398
-	NVM365
-	NVM141
-	NVM638
-	NVM386
-	NVM751
-	NVM609
-	NVM662
-	NVM182
-	NVM337
-	NVM669
-	NVM340
-	NVM169
-	NVM244
-	NVM193
-	NVM021
-	NVM069
-	NVM353

-	NVM692
-	NVM749
-	NVM676
-	NVM594
-	NVM516
-	NVM179
-	NVM278
-	NVM317
-	NVM362
-	NVM613
-	NVM203
-	NVM000
-	NVM252
-	NVM629
-	NVM083
-	NVM713
BSW00302	NVM744
BSW00304	NVM744
BSW00306	NVM744
BSW00307	NVM744
BSW00308	NVM744
BSW00309	NVM744
BSW00312	NVM744
BSW00314	NVM744
BSW00321	NVM744
BSW00323	NVM027
BSW00324	NVM744
BSW00325	NVM744
BSW00326	NVM744
BSW00327	NVM027, NVM023
BSW00328	NVM744
BSW00330	NVM744
BSW00331	NVM027, NVM023
BSW00334	NVM744
BSW00335	NVM744
BSW00336	NVM744
BSW00337	NVM023, NVM024
BSW00338	NVM025
BSW00339	NVM026
BSW00341	NVM744
BSW00342	NVM744
BSW00343	NVM744

BSW00344	NVM744
BSW00347	NVM744
BSW00348	NVM744
BSW00350	NVM025, NVM188
BSW00353	NVM744
BSW00355	NVM744
BSW00361	NVM744
BSW00371	NVM744
BSW00373	NVM464
BSW00375	NVM744
BSW00376	NVM464
BSW00378	NVM744
BSW00380	NVM744
BSW00383	NVM465, NVM466
BSW00384	NVM465, NVM466
BSW00385	NVM027, NVM023
BSW00386	NVM027, NVM025, NVM023
BSW00387	NVM330, NVM331
BSW00398	NVM744
BSW00399	NVM744
BSW004	NVM089
BSW00400	NVM744
BSW00404	NVM744
BSW00405	NVM744
BSW00406	NVM400, NVM399, NVM027, NVM023
BSW00407	NVM286, NVM285
BSW00409	NVM186
BSW00411	NVM286
BSW00412	NVM744
BSW00415	NVM744
BSW00416	NVM744
BSW00417	NVM744
BSW00420	NVM744
BSW00421	NVM465
BSW00422	NVM744
BSW00423	NVM744
BSW00424	NVM322
BSW00425	NVM464
BSW00426	NVM744
BSW00427	NVM744
BSW00428	NVM324
BSW00429	NVM332

BSW00431	NVM744
BSW00432	NVM744
BSW00433	NVM324
BSW00434	NVM744
BSW00435	NVM077
BSW00436	NVM077
BSW00447	NVM755
BSW005	NVM744
BSW006	NVM744
BSW007	NVM744
BSW009	NVM744
BSW010	NVM744
BSW011	NVM157
BSW013	NVM162, NVM699, NVM698
BSW016	NVM010, NVM122, NVM454, NVM051, NVM196, NVM195
BSW017	NVM410, NVM411, NVM122, NVM455, NVM051
BSW018	NVM267, NVM266, NVM012, NVM122, NVM456, NVM051
BSW020	NVM015, NVM451
BSW027	NVM442
BSW030	NVM164
BSW034	NVM162
BSW038	NVM026, NVM191
BSW041	NVM122, NVM051
BSW08000	NVM123, NVM051, NVM442
BSW08007	NVM448
BSW08009	NVM326, NVM325, NVM577
BSW08010	NVM172, NVM171
BSW08011	NVM421, NVM459
BSW08014	NVM122, NVM051, NVM442
BSW08015	NVM397
BSW08528	NVM557, NVM137
BSW08529	NVM558, NVM137
BSW08531	NVM559, NVM137
BSW08533	NVM540, NVM454, NVM460
BSW08534	NVM137
BSW08535	NVM253, NVM018, NVM461
BSW08540	NVM019, NVM458
BSW08541	NVM384, NVM208, NVM472
BSW08542	NVM378, NVM032, NVM564
BSW08544	NVM415, NVM457
BSW08545	NVM405, NVM241, NVM453
BSW08546	NVM548, NVM240

BSW08547	NVM132, NVM165, NVM164, NVM571, NVM174
BSW08548	NVM700
BSW08549	NVM171
BSW08550	NVM345, NVM344, NVM696
BSW08554	NVM526, NVM527, NVM529, NVM213, NVM581
BSW08555	NVM523, NVM524, NVM593
BSW08556	NVM527, NVM528, NVM529
BSW08557	NVM508, NVM507, NVM506
BSW08558	NVM458
BSW08559	NVM535, NVM536
BSW08560	NVM535, NVM536
BSW101	NVM400, NVM399
BSW125	NVM463
BSW127	NVM016, NVM450
BSW129	NVM582, NVM165
BSW130	NVM744
BSW160	NVM744
BSW161	NVM744
BSW162	NVM744
BSW164	NVM744
BSW168	NVM744
BSW170	NVM744
BSW172	NVM324, NVM464
BSW176	NVM540, NVM454, NVM455, NVM461, NVM460

7 Functional specification

7.1 Basic architecture guidelines

7.1.1 Layer structure

The figure below shows the communication interaction of module NvM.

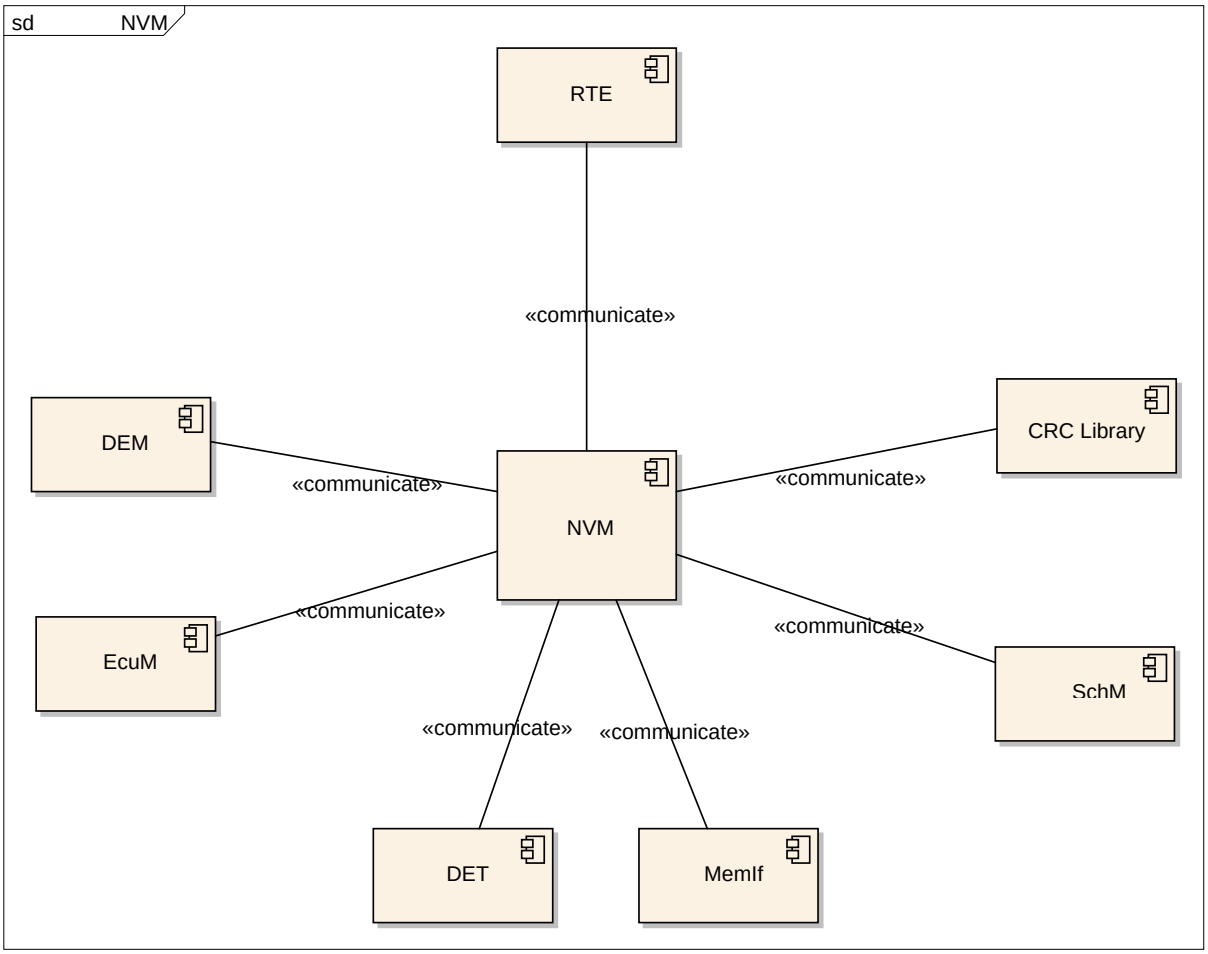

Figure 4: NVRAM Manager interactions overview

7.1.2 Addressing scheme for the memory hardware abstraction

[NVM051] ⌈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 logical block number and a 16bit block address offset.⌋ (BSW041, BSW08000, BSW08014, BSW016, BSW017, BSW018)

Hint: According to [NVM051, the NvM module allows for a (theoretical) maximum of 65536 logical blocks, each logical block having a (theoretical) maximum size of 64 Kbytes.

[NVM122] ⌈The NvM module shall further subdivide the 16bit logical block number into the following parts:

- block identifier with the size of (16 - NvMDatasetSelectionBits)bits
- NvMDatasetSelectionBits bit data index, allowing for up to 256 datasets per NVRAM block⌋(BSW041, BSW08014, BSW016, BSW017, BSW018)

[NVM343] ⌈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. ⌋( )

[NVM123] ⌈The NvM module shall store the block identifier in the most significant bits of the 16bit logical block number. ⌋(BSW08000)

[NVM442] ⌈The configuration tool shall configure the block identifiers. ⌋(BSW08000, BSW027, BSW08014)

[NVM443] ⌈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 NvMNvBlockBaseNumber, NvMDatasetSelectionBits on NVRAM manager side and EA_BLOCK_NUMBER / FEE_BLOCK_NUMBER on memory hardware abstraction side [NVM061_Conf].
For the given examples A and B a simple formula is used:

```
FEE/EA_BLOCK_NUMBER = (NvMNvBlockBaseNumber << NvMDatasetSelectionBits)
+ 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

[NVM125] ⌈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.

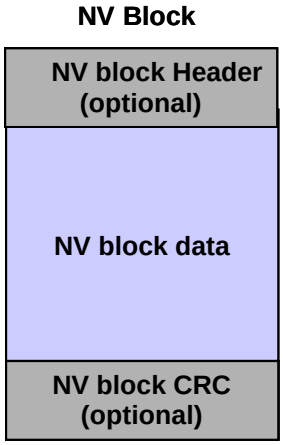

Figure 5: NV Block layout

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

[NVM126] ⌈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. ⌋( )

[NVM127] ⌈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). [NVM061_Conf]. ⌋( )

[NVM129] ⌈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. ⌋( )

[NVM130] ⌈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).

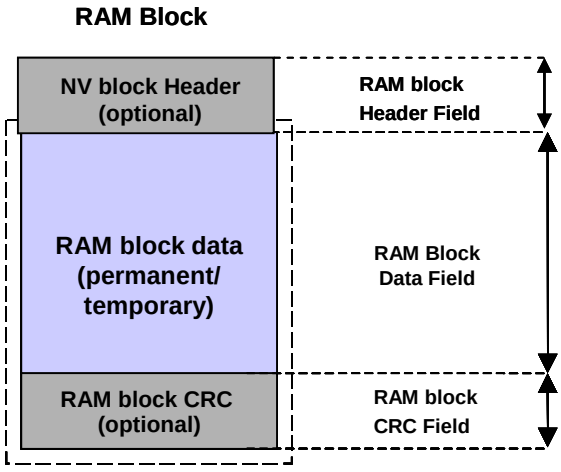

Figure 6: RAM Block layout

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. ⌋( )

[NVM373] ⌈The RAM block data shall contain the permanently or temporarily assigned user data. ⌋( )

[NVM370] ⌈In case of permanently assigned user data, the address of the RAM block data is known during configuration time.⌋ ( )

[NVM372] ⌈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.⌋ ( )

[NVM088] ⌈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

[NVM020] ⌈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.

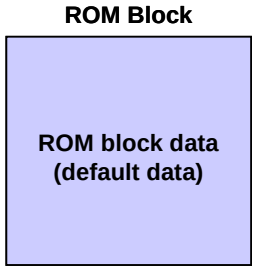

Figure 7: ROM block layout

⌋ ( )

7.1.3.4 Administrative block

[NVM134] ⌈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.⌋ ( )

[NVM128] ⌈The NvM module shall use state information of the permanent RAM block (invalid/valid) to determine the validity of the permanent RAM block user data.⌋ ( )

[NVM132] ⌈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.⌋ (BSW08547)

[NVM133] ⌈The value of “invalid” shall be represented by all other values except “valid”.⌋( )

[NVM135] ⌈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.⌋( )

[NVM054] ⌈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.⌋( )

[NVM136] ⌈The NvM module shall use an error/status field to manage the error/status value of the last request [NVM083].⌋( )

7.1.3.5 NV Block Header

[NVM522] ⌈The NV Block header shall be included first in the NV Block, if the mechanism Static Block ID is enabled.

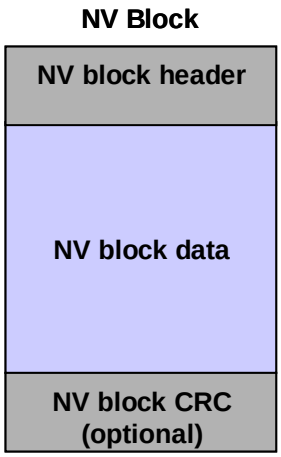

Figure 8: NV block layout with Static Block ID enabled

⌋( )

7.1.4 Block management types

7.1.4.1 Block management types overview

[NVM137] ⌈The following types of NVRAM storage shall be supported by the NvM module implementation:
- NVM_BLOCK_NATIVE
- NVM_BLOCK_REDUNDANT
- NVM_BLOCK_DATASET⌋(BSW08534, BSW08528, BSW08529, BSW08531)

[NVM557] ⌈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⌋(BSW08528)

[NVM558] ⌈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⌋(BSW08529)

[NVM559] ⌈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 NvMDatasetSelectionBits. ⌋(BSW08531)

7.1.4.2 NVRAM block structure

[NVM138] ⌈The NVRAM block shall consist of the mandatory basic storage objects NV block, RAM block and Administrative block. ⌋( )

[NVM139] ⌈The basic storage object ROM block is optional. ⌋( )

[NVM140] ⌈The composition of any NVRAM block is fixed during configuration by the corresponding NVRAM block descriptor. ⌋( )

[NVM141] ⌈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

[NVM069] ⌈A single NVRAM block to deal with will be selected via the NvM module API by providing a subsequently assigned Block ID.⌋( )

[NVM143] ⌈All structures related to the NVRAM block descriptor table and their addresses in ROM (FLASH) have to be generated during configuration of the NvM 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.

[NVM000] ⌈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.

[NVM001] ⌈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:

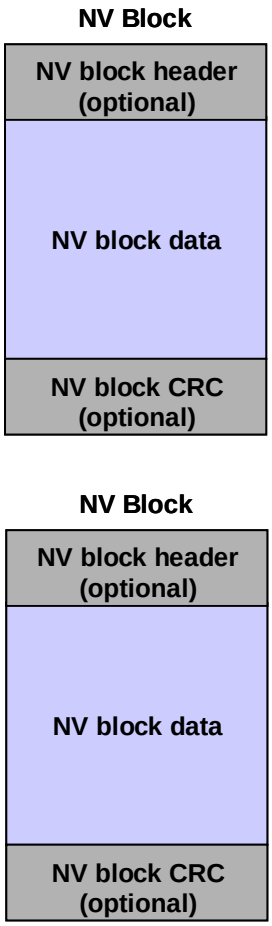

Figure 9: Redundant NVRAM Block layout

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

[NVM531] ⌈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. ⌋( )

[NVM546] ⌈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 can be done immediately or at a later point in time. ⌋( )

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.

[NVM006] ⌈The Dataset NVRAM block consists of multiple NV user data, (optionally) CRC areas, (optional) NV block headers, a RAM block and an Administrative block. ⌋ ( )

[NVM144] ⌈The index position of the dataset is noticed via a separated field in the corresponding Administrative block. ⌋ ( )

[NVM374] ⌈The NvM module shall be able to read all assigned NV blocks. ⌋ ( )

[NVM375] ⌈The NvM module shall only be able to write to all assigned NV blocks if (and only if) write protection is disabled. ⌋ ( )

[NVM146] ⌈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. ⌋ ( )

[NVM376] ⌈The NvM module shall be able to only read optional ROM blocks (default datasets). ⌋ ( )

[NVM377] ⌈The NvM module shall treat a write to a ROM block like a write to a protected NV block. ⌋ ( )

[NVM444] ⌈The total number of configured datasets (NV+ROM blocks) must be in the range of 1..255. ⌋ ( )

[NVM445] ⌈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 + NvMRomBlockNum - 1 represent the ROM blocks.

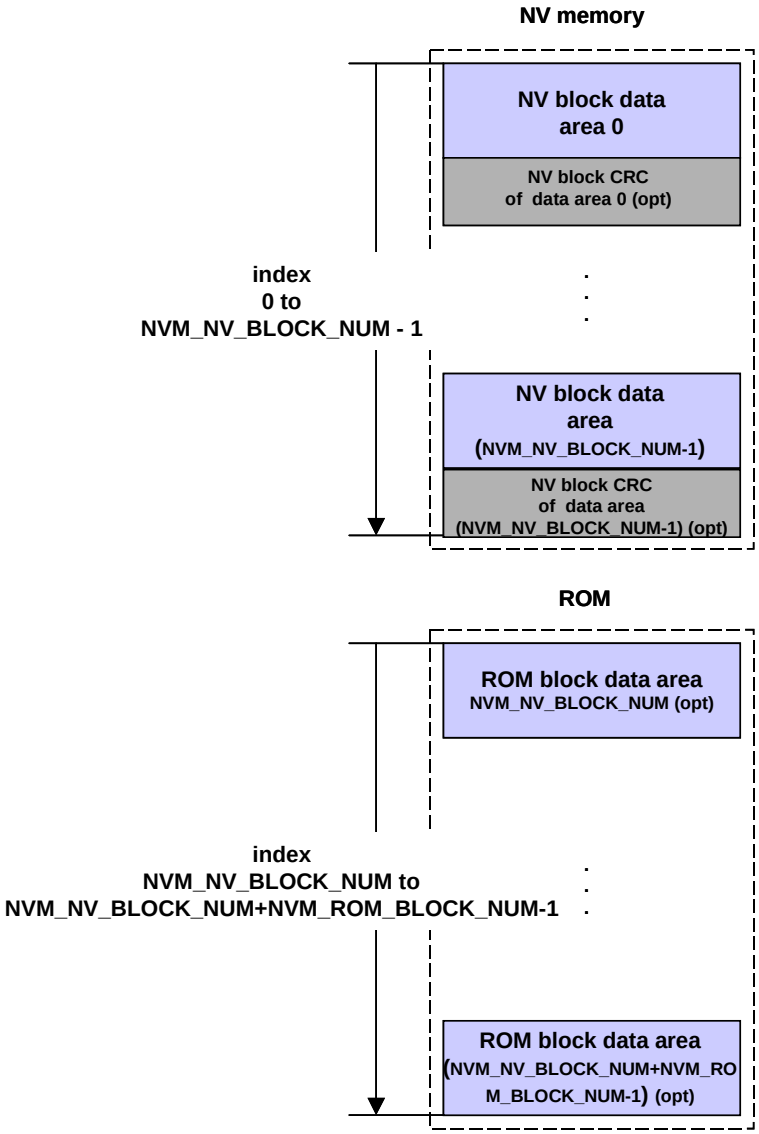

Figure 10: Dataset NVRAM block layout

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

[NVM149] ⌈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. ⌋( )

[NVM560] ⌈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(…)

Type 3:
- NvM_ReadAll(...)
- NvM_WriteAll(...)
- NvM_CancelWriteAll(...)

Type 4:
- NvM_Init(...)⌋ ( )

[NVM561] ⌈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(…)

Type 3:
- NvM_ReadAll(...)
- NvM_WriteAll(...)
- NvM_CancelWriteAll(...)

Type 4:
- NvM_Init(...)⌋ ( )

[NVM562] ⌈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. ⌋( )

[NVM365] ⌈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 8.7.

[NVM150] ⌈The NvM module shall only contain that code that is needed to handle the configured block types. ⌋( )

7.1.5 Scan order / priority scheme

[NVM032] ⌈The NvM module shall support a priority based job processing. ⌋ (BSW08542)

[NVM564] ⌈By configuration parameter NvMJobPrioritization [NVM028] priority based job processing shall be enabled/disabled. ⌋(BSW08542)

[NVM378] ⌈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). ⌋(BSW08542)

[NVM379] ⌈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. ⌋( )

[NVM380] ⌈The job queue length for multi block requests originating from the NvM_ReadAll and NvM_WriteAll shall be one (only one job is queued). ⌋( )

[NVM381] ⌈The NvM module shall not interrupt jobs originating from the NvM_ReadAll request by other requests. ⌋( )

Note: The only exception to the rule given in [NVM381, NVM567] is a write job with immediate priority which shall preempt the running read / write job [NVM182]. The preempted job shall subsequently be resumed / restarted by the NvM module.

[NVM567] ⌈The NvM module shall not interrupt jobs originating from the NvM_WriteAll request by other requests.⌋( )

[NVM568] ⌈The NvM module shall rather queue read jobs that are requested during an ongoing NvM_ReadAll request and executed them subsequently.⌋( )

[NVM569] ⌈The NvM module shall rather queue write jobs that are requested during an ongoing NvM_WriteAll request and executed them subsequently.⌋( )

[NVM725] ⌈The NvM module shall rather queue write jobs that are requested during an ongoing NvM_ReadAll request and executed them subsequently.⌋( )

[NVM726] ⌈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 [[NVM23].
Hint: It shall be allowed to dequeue requests, if they became obsolete by completion of the regarding NVRAM block.

[NVM570] ⌈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

[NVM383] ⌈For each asynchronous request, a notification of the caller after completion of the job shall be a configurable option.⌋( )

[NVM384] ⌈The NvM module shall provide a callback interface [[NVM11].
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.⌋(BSW08541)

[NVM038] ⌈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.⌋( )

[NVM692] ⌈The application accesses the RAM data directly, with respect to given restrictions (e.g. synchronization). ⌋( )

[NVM385] ⌈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). ⌋( )

[NVM386] ⌈The NvM module shall accept multiple asynchronous “single block” requests as long as no queue overflow occurs. ⌋( )

[NVM155] ⌈The highest priority request shall be fetched from the queues by the NvM module and processed in a serialized order. ⌋( )

[NVM040] ⌈The NvM module shall implement implicit mechanisms for consistency / integrity checks of data saved in NV memory [[NVM165]. ⌋( )

[NVM156] ⌈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. ⌋( )

[NVM085] ⌈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 [NVM061_Conf] between first initialization and corrupted data [[NVM08]. ⌋( )

[NVM387] ⌈During processing of NvM_ReadAll, the NvM module shall be able to detect corrupted RAM data by performing a checksum calculation. [NVM476_Conf]. ⌋( )

[NVM226] ⌈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 [NVM476_Conf]. ⌋( )

[NVM388] ⌈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. ⌋( )

[NVM332] ⌈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. ⌋(BSW00429)

7.2.2 Design notes

7.2.2.1 NVRAM manager startup

[NVM693] ⌈NvM_Init shall be invoked by the ECU state manager exclusively. ⌋( )

[NVM091] ⌈Due to strong constraints concerning the ECU startup time, the NvM_Init request shall not contain the initialization of the configured NVRAM blocks. ⌋( )

[NVM157] ⌈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 ECU state manager. ⌋(BSW011)

[NVM158] ⌈The initialization of the RAM data blocks shall be done by another request, namely NvM_ReadAll. ⌋( )

NvM_ReadAll shall be called exclusively by the ECU state manager if EcuM Fixed is used or by integration code if EcuM Flex is used.

[NVM694] ⌈Software components which use the NvM module shall be responsible for checking global error/status information resulting from the NvM module startup. The ECU state manager shall use polling by using NvM_GetErrorStatus [[NVM01] (reserved block ID 0) or callback notification (configurable option NvM_MultiBlockCallback [NVM028]) 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) [[NVM08]. If callbacks are chosen for notification, software components shall be notified automatically if an assigned NVRAM block has been processed [[NVM28].

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. ⌋( )

[NVM160] ⌈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 [[NVM01] with specific block IDs to determine the validity of the corresponding RAM blocks. ⌋( )

[NVM695] ⌈For all blocks of the block management type “NVRAM Dataset” [[NVM006] the software component shall be responsible to set the proper index position by NvM_SetDataIndex [[NVM01]. 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 [[NVM02] API call. ⌋( )

7.2.2.2 NVRAM manager shutdown

[NVM092] ⌈The basic shutdown procedure shall be done by the request NvM_WriteAll [[NVM01].
Hint: NvM_WriteAll shall be invoked by the ECU state manager. ⌋( )

7.2.2.3 (Quasi) parallel write access to the NvM module

[NVM162] ⌈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. ⌋(BSW013, BSW034)

7.2.2.4 NVRAM block consistency check

[NVM164] ⌈The NvM module shall provide implicit techniques to check the data consistency of NVRAM blocks [NVM476_Conf], [[NVM040]. ⌋(BSW08547, BSW030)

[NVM571] ⌈The data consistency check of a NVRAM block shall be done by CRC recalculations of its corresponding NV block(s). ⌋(BSW08547)

[NVM165] ⌈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 `NvMBlockUseCrc` and `NvMCalcRamBlockCrc` [NVM061_Conf]. ⌋(BSW08547, BSW129)

[NVM724] ⌈`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. ⌋( )

[NVM544] ⌈Depending on the configurable parameters `NvMBlockUseCrc` and `NvMCalcRamBlockCrc`, NvM module shall allocate memory for the largest CRC used.
Hint: NvM users must not know anything about CRC memory (e.g. size, location) for their data in a RAM block.⌋( )

7.2.2.5 Error recovery

[NVM047] ⌈The NvM module shall provide techniques for error recovery. The error recovery depends on the NVRAM block management type [[NVM001].⌋( )

[NVM389] ⌈The NvM module shall provide error recovery on read for every kind of NVRAM block management type by loading of default values.⌋( )

[NVM390] ⌈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.⌋( )

[NVM168] ⌈The NvM module shall provide error recovery on write by performing write retries regardless of the NVRAM block management type.⌋( )

[NVM169] ⌈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

[NVM171] ⌈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 [[NVM387**,** **[NVM226,**[NVM388]. ⌋ (BSW08549, BSW08010)

Application hint:

As the NvM module does not provide a mechanism or special status information to inform the caller that a ROM block has been loaded due to recovery of a RAM block, this has to be managed by e.g. SW-Cs itself. A possible solution could be to add additional data marking the block as ROM defaults.

7.2.2.7 Implicit recovery of a RAM block with ROM default data

[NVM172] ⌈The data content of the corresponding NV block shall remain unmodified during the implicit recovery.⌋(BSW08010)

[NVM572] ⌈The implicit recovery shall not be provided during startup (part of NvM_ReadAll) and NvM_ReadBlock for each NVRAM block when no ROM block is configured.⌋( )

[NVM573] ⌈The implicit recovery shall not be provided during startup (part of NvM_ReadAll) and NvM_ReadBlock for each NVRAM block for the following conditions:

- The ROM block is configured.
- The permanent RAM block state is valid and CRC (data) is consistent.⌋( )

[NVM574] ⌈The implicit recovery shall not be provided during startup (part of NvM_ReadAll) and NvM_ReadBlock for each NVRAM block for the following conditions:

- The ROM block is configured.
- The permanent RAM block state is invalid and CRC (data) is inconsistent.
- Read attempt from NV success.⌋( )

[NVM575] ⌈The implicit recovery shall be provided during startup (part of NvM_ReadAll) and NvM_ReadBlock for each NVRAM block for the following conditions:

- The ROM block is configured.
- The permanent RAM block state is invalid and CRC (data) is inconsistent.
- Read attempt from NV fails.⌋( )

7.2.2.8 Explicit recovery of a RAM block with ROM default data

[NVM391] ⌈For explicit recovery with ROM block data the NvM module shall provide a function NvM_RestoreBlockDefaults [[NVM01] to restore ROM data to its corresponding RAM block.⌋( )

[NVM392] ⌈The function NvM_RestoreBlockDefaults shall remain unmodified the data content of the corresponding NV block.
Hint: The function NvM_RestoreBlockDefaults 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

[NVM174] ⌈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. ⌋(BSW08547)

7.2.2.10 Termination of a single block request

[NVM175] ⌈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 [[NVM000]. ⌋( )

[NVM176] ⌈The optional configuration parameter NvMSingleBlockCallback configures the notification via callback on the termination of an asynchronous block request (except for NvM_CancelWriteAll) [NVM061_Conf].

Note: In communication with application SW-C, the NvMSingleBlockCallback shall be mapped to the `Rte_call_<p>_<o>` API. ⌋( )

7.2.2.11 Termination of a multi block request

[NVM393] ⌈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`). ⌋( )

[NVM394] ⌈The function `NvM_GetErrorStatus` [[NVM01] shall return the most recent error/status information of an asynchronous multi block request (including `NvM_CancelWriteAll`) [[NVM08] in conjunction with a reserved block ID value of 0. ⌋( )

[NVM395] ⌈The result of a multi block request shall represent only a common error/status information. ⌋( )

[NVM396] ⌈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. ⌋( )

[NVM179] ⌈The optional configuration parameter NvMMultiBlockCallback configures the notification via callback on the termination of an asynchronous multi block request [NVM028]. ⌋( )

7.2.2.12 General handling of asynchronous requests/ job processing

[NVM180] ⌈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 NvMCrcNumOfBytes.⌋( )

[NVM351] ⌈For CRC calculation, the NvM module shall use initial values which are published by the CRC module.⌋( )

[NVM181] ⌈Multiple concurrent single block requests shall be queueable.⌋( )

[NVM182] ⌈The NvM module shall interrupt asynchronous request/job processing in favor of jobs with immediate priority (crash data).⌋( )

[NVM184] ⌈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`.⌋( )

[NVM185] ⌈On successful enqueuing a request, the NvM module shall set the request result of the corresponding NVRAM block to `NVM_REQ_PENDING`.⌋( )

[NVM270] ⌈If the NvM module has successfully processed a job, it shall return NVM_REQ_OK as job 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.

[NVM325] ⌈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.⌋(BSW08009)

[NVM577] ⌈Enabling/Disabling of the write protection is not allowed using NvM_SetBlockProtection function when the NvMWriteBlockOnce is TRUE regardless of the value (True/False) configured for NvMBlockWriteProt.⌋(BSW08009)

[NVM326] ⌈For all NVRAM blocks configured with NvMBlockWriteProt = TRUE, the NvM module shall enable a default write protection.⌋(BSW08009)

[NVM578] ⌈The NvM module's environment can explicitly disable the write protection using the NvM_SetBlockProtection function.⌋( )

[NVM397] ⌈For NVRAM blocks configured with `NvMWriteBlockOnce == TRUE` [NVM072], the NvM module shall only write once to the associated NV memory, i.e in case of a blank NV device. ⌋(BSW08015)

[NVM398] ⌈For NVRAM blocks configured with `NvMWriteBlockOnce == TRUE`, the NvM module shall not allow disabling the write protection explicitly using the NvM_SetBlockProtection function.[NVM450] ⌋( )

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. The following figures depict the state transitions of RAM blocks.

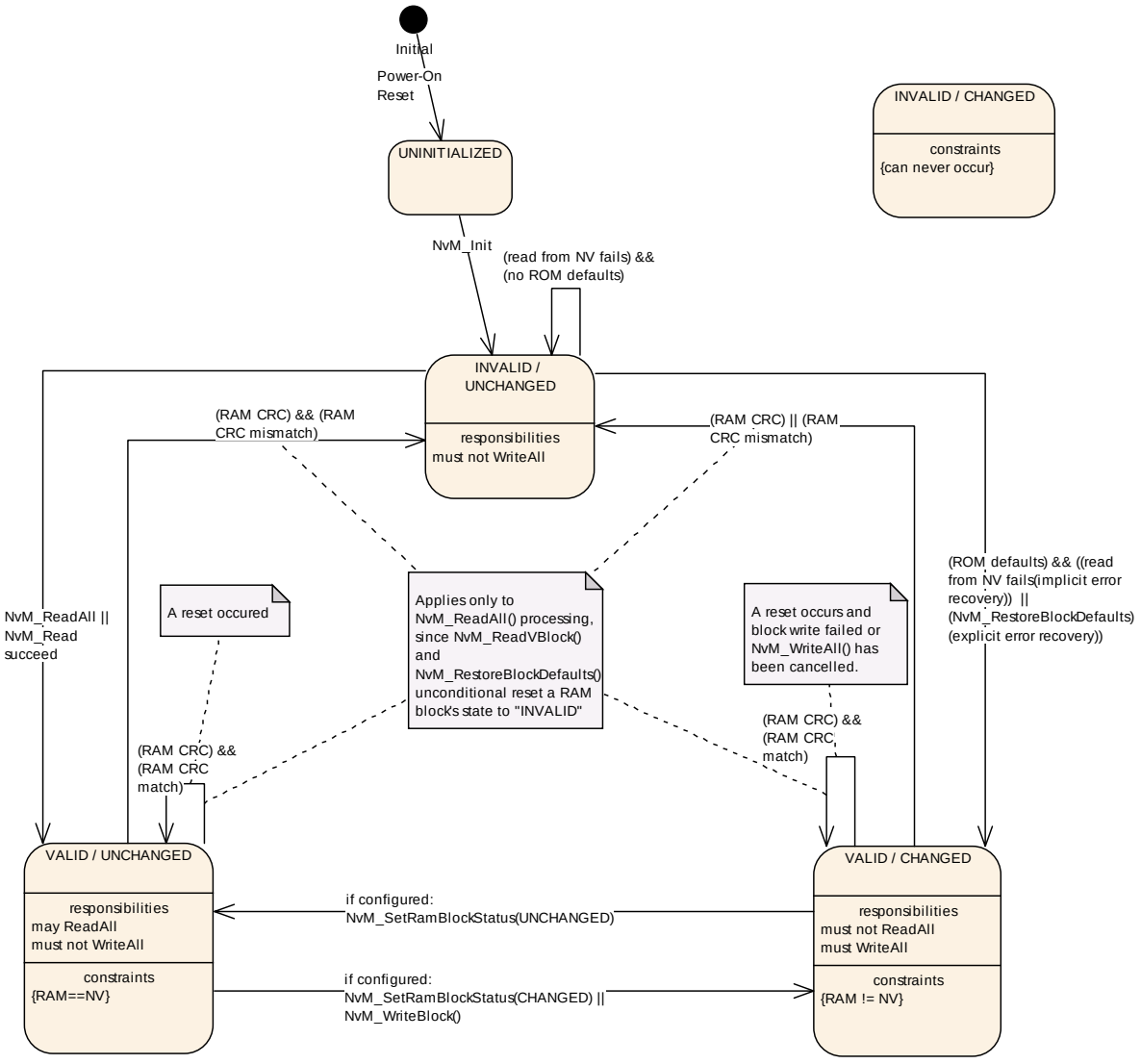

Figure 11: RAM Block States

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.

[NVM344] ⌈If the API for modifying the RAM block status has been configured out (via NvMSetRamBlockStatusApi) the NvM module shall treat a RAM block as valid and modified when writing to it, i.e. during `NvM_WriteAll`, the NvM module shall write each permanent RAM block to NV memory. ⌋(BSW08550)

[NVM345] ⌈If the API for modifying the RAM block status has been configured out (via NvMSetRamBlockStatusApi) the NvM module shall treat a RAM block as invalid when reading it, i.e. during NvM_ReadAll, the NvM module shall copy each NVRAM block to RAM if configured accordingly. ⌋(BSW08550)

[NVM696] ⌈In case of an unsuccessful block read attempt, it is the responsibility of the application to provide valid data before the next write attempt. ⌋(BSW08550)

[NVM472] ⌈In case a RAM block is successfully copied to NV memory the RAM block state shall be set to "valid/unmodified" afterwards.
⌋(BSW08541)

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 must 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.

[NVM697] ⌈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)

[NVM698] ⌈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 request which transfers control to the NvM module.
3. From now on the application must 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.⌋(BSW013)

7.2.2.15.2 Read requests (NvM_ReadBlock)

[NVM699] ⌈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 must 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.⌋(BSW013)

7.2.2.15.3 Restore default requests (NvM_RestoreBlockDefaults)

[NVM700] ⌈Applications have to adhere to the following rules during restore default requests for implicit synchronization between application and NVRAM manager:

6. The application provides a RAM block, which has to be filled with ROM data from the NvM modules side.

7. The application issues the NvM_RestoreBlockDefaults request which transfers control to the NvM module.
8. From now on the application must not read or write to the RAM block until success or failure of the request is signaled or derived via polling.
9. An application can use polling to get the status of the request or can be informed via a callback function.
10. After completion of the NvM module operation, the RAM block is available with the ROM data for use by the application. 」(BSW08548)

7.2.2.15.4 Multi block read requests (NvM_ReadAll)

This request may be triggered only by the ECU state manager if EcuM Fixed is used or by integration code if EcuM Flex is used 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 ECU state manager. The DEM and the ECU state 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 ECU state manager.

[NVM701] 「Applications have to adhere to the following rules during multi block read requests for implicit synchronization between application and NVRAM manager:
The ECU state manager issues the NvM_ReadAll.

1. The ECU state 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 must only be triggered by the ECU state 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 ECU state 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 ECU state manager.

[NVM702] 「Applications have to adhere to the following rules during multi block write requests for implicit synchronization between application and NVRAM manager:

1. The ECU state manager issues the NvM_WriteAll request which transfers control to the NvM module.
2. The ECU state 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.

[NVM703] ⌈NvM_CancelWriteAll request shall only be used by the ECU state 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).

[NVM704] ⌈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 [NVM034) 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.

[NVM511] ⌈For every NVRAM block there shall be the possibility to configure the usage of an explicit synchronization mechanism by the parameter NvMBlockUseSyncMechanism.⌋( )

[NVM512] ⌈The NvM module must not allocate a RAM mirror if no block is configured to use the explicit synchronization mechanism.⌋( )

[NVM513] ⌈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 must not exceed the size of the longest NVRAM block configured to use the explicit synchronization mechanism.⌋( )

[NVM514] ⌈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 NvMBlockUseSyncMechanism == TRUE. The buffer must not be used for the other NVRAM blocks.⌋( )

[NVM515] ⌈The NvM module shall call the routine NvMWriteRamBlockToNvM in order to copy the data from the RAM block to the mirror for all NVRAM blocks with NvMBlockUseSyncMechanism == TRUE. This routine must not be used for the other NVRAM blocks.⌋( )

[NVM516] ⌈The NvM module shall call the routine NvMReadRamBlockFromNvM in order to copy the data from the mirror to the RAM block for all NVRAM blocks with NvMBlockUseSyncMechanism == TRUE. This routine must not be used for the other NVRAM blocks.⌋( )

[NVM517] ⌈If the routines NvMReadRamBlockFromNvM return E_NOT_OK, then the NvM module shall retry the routine call NvMRepeatMirrorOperations times. Thereafter the NvM module shall process the next job in the queue if available and then restart processing the previous job.⌋( )

[NVM579] ⌈If the routines NvMWriteRamBlockToNvM return E_NOT_OK, then the NvM module shall retry the routine call NvMRepeatMirrorOperations times. Thereafter the NvM module shall process the next job in the queue if available and then restart processing the previous job. ⌋( )

The following two sections clarify the differences when using the explicit synchronization mechanism, compare to 7.2.2.15.1 and 7.2.2.15.2.

7.2.2.17.1 Write requests (NvM_WriteBlock)

[NVM705] ⌈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 request.
3. The application might modify the RAM block until the routine NvMWriteRamBlockToNvM is called by the NvM module.
4. If the routine NvMWriteRamBlockToNvM 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.
8. Note: The application may combine several write requests to different positions in one RAM block, if NvM_WriteBlock was requested, but not yet processed by the NvM module. The request was not processed, if the callback routine NvMWriteRamBlockToNvM was not called. ⌋( )

7.2.2.17.2 Read requests (NvM_ReadBlock)

[NVM706] ⌈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 request.
3. The application might modify the RAM block until the routine NvMReadRamBlockFromNvM is called by the NvM module.
4. If the routine NvMReadRamBlockFromNvM 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

NvMRepeatMirrorOperations 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 was requested, but not yet processed by the NvM module. The request was not processed, if the callback routine NvMReadRamBlockFromNvM was not called.

Note: NvM_RestoreBlockDefaults works similarly to NvM_ReadBlock.」( )

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.

[NVM523] ⌈The NVRAM Manager shall store the Static Block ID field of the Block Header each time the block is written to NV memory.⌋(BSW08555)

[NVM524] ⌈The NVRAM Manager shall check the Block Header each time the block is read from NV memory.⌋(BSW08555)

[NVM525] ⌈If the Static Block ID check fails then the failure NVM_E_WRONG_BLOCK_ID is reported to DEM.⌋( )

[NVM580] ⌈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

[NVM526] ⌈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.⌋(BSW08554)

[NVM581] ⌈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.⌋(BSW08554)

[NVM582] ⌈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.⌋(BSW129)

[NVM583] ⌈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.

[NVM527] ⌈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.⌋(BSW08554, BSW08556)

[NVM528] ⌈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.⌋(BSW08556)

[NVM529] ⌈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.⌋(BSW08554, BSW08556)

[NVM530] ⌈If the read back operation fails then no read retries shall be performed.⌋( )

7.2.2.21 NvM and BswM interaction

[NVM745] ⌈If BswMMultiBlockJobStatusInformation is true the NVM shall inform the BSWM about the current state of a multi block job via BswM_NvM_CurrentJobMode and the configured multi job callback should not be called.⌋( )

[NVM746] ⌈If BswMBlockStatusInformation is true, the NVM shall inform the BSWM about the current state of the block via BswM_NvM_CurrentBlockMode.⌋( )

7.2.2.22 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.

[NVM751] ⌈ If the API was called with parameter Locked 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_InvalidateNvBlock, NvM_EraseNvBlock, shall be rejected. ⌋ ( )

[NVM752] ⌈ If the API was called with parameter Locked as TRUE, the NVM shall guarantee that at next start-up, during processing of NvM_ReadBlock, this NVRAM block shall be loaded from NV memory. ⌋ ( )

[NVM753] ⌈If the Locked parameter got the value FALSE, the NVM shall guarantee normal processing of this NVRAM block as specified by AUTOSAR. ⌋ ( )

[NVM754] ⌈The setting made using this service shall not be changeable by NvM_SetRamBlockStatus, nor by NvM_SetBlockProtection. ⌋ ( )

7.2.2.22.1 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`).

7.2.2.22.2 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(<BlockId>, TRUE)`.

7.3 Error classification

[NVM186] ⌈Values for production code Event Ids are assigned externally by the configuration of the Dem. ⌋(BSW00409)

[NVM585] ⌈Values for production code Event Ids are published in the file Dem_IntErrId.h. ⌋( )

[NVM739] ⌈If not specified in a special case differently: For all production errors reported to the DEM the EventStatus shall be set to DEM_EVENT_STATUS_FAILED. ⌋( )

[NVM187] ⌈Development error values are of type uint8. ⌋( )

[NVM023] ⌈The Development errors
- NVM_E_PARAM_BLOCK_ID (0x0A)
- NVM_E_PARAM_BLOCK_TYPE (0x0B)
- NVM_E_PARAM_BLOCK_DATA_IDX (0x0C)
- NVM_E_PARAM_ADDRESS (0x0D)
- NVM_E_PARAM_DATA (0x0E)
- NVM_E_PARAM_POINTER (0x0F)

shall be detectable by the NvM module when API requests are called with wrong parameters, depending on whether the build version mode is development mode. ⌋ (BSW00385, BSW00386, BSW00406, BSW00337, BSW00327, BSW00331)

[NVM586] ⌈The Development error NVM_E_NOT_INITIALIZED (0x14) shall be detectable by the NvM module when NVRAM manager is still not initialized, depending on whether the build version mode is development mode. ⌋( )

[NVM587] ⌈The Development error NVM_E_BLOCK_PENDING (0x15) shall be detectable by the NvM module when API read/write/control request failed because a block with the same ID is already listed or currently in progress, depending on whether the build version mode is development mode. ⌋( )

[NVM590] ⌈The development error NVM_E_BLOCK_CONFIG (0x18) shall be detectable by the NvM module when the service is not possible with this block configuration, depending on whether the build version mode is development mode. ⌋( )

[NVM747] ⌈The development error NVM_E_BLOCK_LOCKED (0x19) shall be detectable by the NvM module when API write request failed for this block because RAM block is locked, depending on whether the build version mode is development mode. ⌋( )

[NVM591] ⌈The 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.⌋( )

[NVM592] ⌈The 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.⌋( )

[NVM593] ⌈The 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.⌋(BSW08555)

[NVM594] ⌈The 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.⌋( )

[NVM595] ⌈The 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.⌋( )

[NVM722] ⌈The Production error NVM_E_QUEUE_OVERFLOW (value assigned by DEM, see container NvmDemEventParameterRefs) shall be detectable by the NvM module when an NVRAM Managers job queue overflow occurs.⌋( )

[NVM723] ⌈The Production error NVM_E_WRITE_PROTECTED (value assigned by DEM) shall be detectable by the NvM module when a write attempt to s NVRAM block with write protection occurs.⌋( )

[NVM024] ⌈Additional errors that are detected because of specific implementation and/or specific hardware properties shall be added in the NvM module implementation specification. The classification and enumeration shall be compatible to the errors listed above [[NVM023].⌋(BSW00337)

7.4 Error detection

[NVM025] ⌈The detection of development errors shall be pre compile time configurable by the configuration parameter: NvMDevErrorDetect.⌋(BSW00386, BSW00338, BSW00350)

[NVM596] ⌈The detection of development errors shall be configurable On/Off by the configuration parameter: NvMDevErrorDetect. ⌋( )

[NVM597] ⌈The switch NvMDevErrorDetect shall activate or deactivate the detection of all development errors.

The development errors are notified as described in Section 7.5. ⌋( )

[NVM188] ⌈If the NvMDevErrorDetect switch is enabled API parameter checking is enabled. The detailed description of the detected errors can be found in chapter 7.3. ⌋(BSW00350)

[NVM189] ⌈The detection of production code errors cannot be switched off. ⌋( )

[NVM027] ⌈If development error detection is enabled for NvM module, the function NvM_SetDataIndex shall report the DET error NVM_E_NOT_INITIALIZED when NVM is not yet initialized. ⌋(BSW00323, BSW00385, BSW00386, BSW00406, BSW00327, BSW00331)

[NVM598] ⌈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. ⌋( )

[NVM599] ⌈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 [NVM444, [NVM445. ⌋( )

[NVM600] ⌈If development error detection is enabled for NvM module, the function NvM_SetDataIndex shall report the DET error NVM_E_PARAM_BLOCK_TYPE when the request is not possible in conjunction with the configured block management type. ⌋( )

[NVM601] ⌈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. ⌋( )

[NVM602] ⌈If development error detection is enabled for NvM module, the function NvM_GetDataIndex shall report the DET error NVM_E_NOT_INITIALIZED when NVM not yet initialized. ⌋( )

[NVM603] ⌈If development error detection is enabled for NvM module, the function NvM_GetDataIndex shall report the DET error NVM_E_PARAM_BLOCK_TYPE

when the request is not possible in conjunction with the configured block management type.」( )

[NVM604] 「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.」( )

[NVM605] 「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.」( )

[NVM606] 「If development error detection is enabled for NvM module, the function NvM_SetBlockProtection shall report the DET error NVM_E_NOT_INITIALIZED when NVM is not yet initialized.」( )

[NVM607] 「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.」( )

[NVM608] 「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.」( )

[NVM609] 「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.」( )

[NVM610] 「If development error detection is enabled for NvM module, the function NvM_GetErrorStatus shall report the DET error NVM_E_NOT_INITIALIZED when NVM is not yet initialized.」( )

[NVM611] 「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.」( )

[NVM612] 「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.」( )

[NVM613] 「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.」( )

[NVM614] ⌈If development error detection is enabled for NvM module, the function NvM_ReadBlock shall report the DET error NVM_E_NOT_INITIALIZED when NVM is not yet initialized.⌋( )

[NVM615] ⌈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.⌋( )

[NVM616] ⌈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 is configured and a NULL pointer is passed via the parameter NvM_DstPtr.⌋( )

[NVM618] ⌈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.⌋( )

[NVM619] ⌈If development error detection is enabled for NvM module, the function NvM_WriteBlock shall report the DET error NVM_E_NOT_INITIALIZED when NVM not yet initialized.⌋( )

[NVM620] ⌈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.⌋( )

[NVM622] ⌈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 is configured and a NULL pointer is passed via the parameter NvM_SrcPtr.⌋( )

[NVM624] ⌈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.⌋( )

[NVM748] ⌈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.⌋( )

[NVM625] ⌈If development error detection is enabled for NvM module, the function NvM_RestoreBlockDefaults shall report the DET error NVM_E_NOT_INITIALIZED when NVM is not yet initialized.⌋( )

[NVM626] ⌈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. ⌋ ( )

[NVM628] ⌈If development error detection is enabled for NvM module, the function NvM_RestoreBlockDefaults shall report the DET error NVM_E_BLOCK_CONFIG when Default data is not available/configured for the referenced NVRAM block. ⌋ ( )

[NVM629] ⌈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 is configured and a NULL pointer is passed via the parameter NvM_DstPtr. ⌋ ( )

[NVM630] ⌈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. ⌋ ( )

[NVM631] ⌈If development error detection is enabled for NvM module, the function NvM_EraseNvBlock shall report the DET error NVM_E_NOT_INITIALIZED when the NVM is not yet initialized. ⌋ ( )

[NVM632] ⌈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. ⌋ ( )

[NVM635] ⌈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. ⌋ ( )

[NVM636] ⌈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. ⌋ ( )

[NVM637] ⌈If development error detection is enabled for NvM module, the function NvM_CancelWriteAll shall report the DET error NVM_E_NOT_INITIALIZED when NVM is not yet initialized. ⌋ ( )

[NVM638] ⌈If development error detection is enabled for NvM module, the function NvM_InvalidateNvBlock shall report the DET error NVM_E_NOT_INITIALIZED when NVM is not yet initialized. ⌋ ( )

[NVM639] ⌈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. ⌋ ( )

[NVM642] ⌈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.⌋( )

[NVM643] ⌈If development error detection is enabled for NvM module, the function NvM_SetRamBlockStatus shall report the DET error NVM_E_NOT_INITIALIZED when NVM not yet initialized.⌋( )

[NVM644] ⌈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.⌋( )

[NVM645] ⌈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.⌋( )

[NVM646] ⌈If development error detection is enabled for NvM module, the function NvM_ReadAll shall report the DET error NVM_E_NOT_INITIALIZED when NVM is not yet initialized.⌋( )

[NVM647] ⌈If development error detection is enabled for NvM module, the function NvM_WriteAll shall report the DET error NVM_E_NOT_INITIALIZED when NVM is not yet initialized.⌋( )

[NVM648] ⌈If development error detection is enabled for NvM module, the function NvM_CancelJobs shall report the DET error NVM_E_NOT_INITIALIZED when NVM is not yet initialized.⌋( )

[NVM649] ⌈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.⌋( )

[NVM728] ⌈If development error detection is enabled for NvM module, the function NvM_SetBlockLockStatus shall report the DET error NVM_E_NOT_INITIALIZED when NVM is not yet initialized.⌋( )

[NVM729] ⌈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.⌋( )

[NVM730] ⌈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.⌋( )

[NVM731] ⌈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.⌋( )

7.5 Error notification

[NVM026] ⌈Production errors shall be reported to the Diagnostic Event Manager.⌋ (BSW00339, BSW038)

[NVM191] ⌈Detected development errors shall be reported to the Det_ReportError service of the Development Error Tracer (DET) if the pre-processor switch NVM_DEV_ERROR_DETECT is set (see chapter 10).⌋(BSW038)

7.6 Version check

[NVM089] ⌈The NvM module shall perform Inter Module Checks to avoid integration of incompatible files. The imported included files shall be checked by preprocessing directives. The following version numbers shall be verified:

- `<MODULENAME>_AR_RELEASE_MAJOR_VERSION`
- `<MODULENAME>_AR_RELEASE_MINOR_VERSION`

Where `<MODULENAME>` is the module short name of the other (external) modules which provide header files included by the NvM module.

If the values are not identical to the expected values, an error shall be reported.⌋ (BSW004)

7.7 Debugging

[NVM506] ⌈Each variable that shall be accessible by AUTOSAR Debugging shall be defined as global variable.⌋(BSW08557)

[NVM507] ⌈All type definitions of variables which shall be debugged shall be accessible by the header file NvM.h.⌋(BSW08557)

[NVM508] ⌈The declaration of variables in the header file shall be such, that it is possible to calculate the size of the variables by C-"sizeof".⌋(BSW08557)

[NVM509] ⌈Variables available for debugging shall be described in the respective Basic Software Module Description.⌋( )

[NVM510] ⌈The internal state “INIT DONE” shall be available for debugging.⌋ ( )

8 API specification

8.1 Imported types

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

[NVM446] ⌈

Module	*Imported Type*
Dem	Dem_EventIdType
	Dem_EventStatusType
MemIf	MemIf_JobResultType
	MemIf_ModeType
	MemIf_StatusType
Std_Types	Std_ReturnType
	Std_VersionInfoType

⌋ ( )

8.2 Type definitions

8.2.1 NvM_RequestResultType

[NVM083] ⌈The type NvM_RequestResultType is an asynchronous request result, which will be returned by the API service NvM_GetErrorStatus [[NVM01]. ⌋ ( )

[NVM470] ⌈

Name:	NvM_RequestResultType		
Type:	uint8		
Range:	NVM_REQ_OK	0x00	0x00: The last asynchronous read/write/control 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	0x01: The last asynchronous read/write/control request has been finished unsuccessfully.
	NVM_REQ_PENDING	0x02	0x02: An asynchronous read/write/control request is currently pending.
	NVM_REQ_INTEGRITY_FAILED	0x03	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	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	0x05: The referenced NV block is invalidated.
	NVM_REQ_CANCELED	0x06	0x06: The multi block request NvM_WriteAll was canceled by calling NvM_CancelWriteAll. Or Any single block job request (NvM_ReadBlock, NvM_WriteBlock, NvM_EraseNvBlock, NvM_InvalidateNvBlock and NvM_RestoreBlockDefaults) was canceled by calling NvM_CancelJobs.
	NVM_REQ_REDUNDANCY_FAILED	0x07	0x07: The required redundancy of the referenced NV block is lost.
	NVM_REQ_RESTORED_FROM_ROM	0x08	0x08: The referenced NV block has been restored from ROM.
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.		

⌋ ( )

8.2.2 NvM_BlockIdType

[NVM471] ⌈

Name:	NvM_BlockIdType		
Type:	uint16		
Range:	0..2^(16-NvMDatasetSelectionBits)-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		

⌋ ( )

[NVM475] ⌈The NVRAM block IDs are expected to be in a sequential order, i.e. the NVRAM manager does not need to be capable of handling non-sequential NVRAM block IDs. ⌋ ( )

8.3 Function definitions

8.3.1 Synchronous requests

8.3.1.1 NvM_Init

[NVM447] ⌈

Service name:	NvM_Init
Syntax:	void NvM_Init(void)
Service ID[hex]:	0x00
Sync/Async:	Synchronous
Reentrancy:	Non Reentrant
Parameters (in):	None
Parameters (inout):	None
Parameters (out):	None
Return value:	None
Description:	Service for resetting all internal variables.

⌋ ( )

[NVM399] ⌈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. ⌋ (BSW101, BSW00406)

[NVM400] ⌈The function NvM_Init shall not modify the permanent RAM block contents, as this shall be done on NvM_ReadAll. ⌋ (BSW101, BSW00406)

[NVM192] ⌈The function NvM_Init shall set the dataset index of all NVRAM blocks of type NVM_BLOCK_DATASET to zero. ⌋ ( )

[NVM193] ⌈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 [NVM028] and published configuration parameter.

Hint: The time consuming NVRAM block initialization and setup according to the block descriptor [NVM061_Conf] shall be done by the NvM_ReadAll request.

8.3.1.2 NvM_SetDataIndex

[NVM448] ⌈

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.	

⌋(BSW08007)

[NVM014] ⌈The function NvM_SetDataIndex shall set the index to access a certain dataset of a NVRAM block (with/without ROM blocks). ⌋( )

[NVM263] ⌈The function NvM_SetDataIndex shall leave the content of the corresponding RAM block unmodified. ⌋( )

[NVM264] ⌈For blocks with block management different from NVM_BLOCK_DATASET, NvM_SetDataIndex shall return without any effect in production mode. ⌋( )

Regarding error detection, the requirement [NVM02 is applicable to the function NvM_SetDataIndex.

[NVM707] ⌈The NvM module's environment shall have initialized the NvM module before it calls the function NvM_SetDataIndex. ⌋( )

Hint: NVRAM common block configuration parameters [NVM028], block management types [NVM061_Conf] and one configured NVRAM block descriptor needed [NVM062].

8.3.1.3 NvM_GetDataIndex

[NVM449] ⌈

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	

⌋ ( )

[NVM021] ⌈The function NvM_GetDataIndex shall get the current index (association) of a dataset NVRAM block (with/without ROM blocks). ⌋ ( )

[NVM265] ⌈For blocks with block management different from NVM_BLOCK_DATASET, NvM_GetDataIndex shall set the index pointed by DataIndexPtr to zero.
Regarding error detection, the requirement [NVM02 is applicable to the function NvM_GetDataIndex. ⌋ ( )

[NVM708] ⌈The NvM module's environment shall have initialized the NvM module before it calls the function NvM_GetDataIndex.
Hint: NVRAM common block configuration parameters [NVM028], block management types [NVM061_Conf] and one configured NVRAM block descriptor needed [NVM062]. ⌋ ( )

8.3.1.4 NvM_SetBlockProtection

[NVM450] ⌈

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.	

⌋(BSW127)

[NVM016] ⌈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.
Regarding error detection, the requirement [NVM02 is applicable to the function NvM_SetBlockProtection. ⌋(BSW127)

[NVM709] ⌈The NvM module's environment shall have initialized the NvM module before it calls the function NvM_SetBlockProtection.
Hint: NVRAM common block configuration parameters [NVM028], block management types [NVM061_Conf] and one configured NVRAM block descriptor needed [NVM062]. ⌋( )

8.3.1.5 NvM_GetErrorStatus

[NVM451] ⌈

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_RequestResultType .
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.	

⌋(BSW020)

[NVM015] ⌈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.
Regarding error detection, the requirement [NVM02 is applicable to the function NvM_GetErrorStatus.⌋(BSW020)

[NVM710] ⌈The NvM module's environment shall have initialized the NvM module before it calls the function NvM_GetErrorStatus.⌋( )

NVRAM common block configuration parameters [NVM028], block management types [NVM061_Conf] and one configured NVRAM block descriptor are needed in the configuration with respect to the function NvM_GetErrorStatus [NVM062].

8.3.1.6 NvM_GetVersionInfo

[NVM452] ⌈

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.

⌋ ( )

[NVM285] ⌈The function NvM_GetVersionInfo shall return the version information of this module. The version information includes:

- Module Id
- Vendor Id
- Vendor specific version numbers (BSW00407). ⌋ (BSW00407)

[NVM286] ⌈The function NvM_GetVersionInfo shall be pre compile time configurable by the configuration parameter NvMVersionInfoApi. ⌋ (BSW00407, BSW00411)

[NVM650] ⌈The function NvM_GetVersionInfo shall be configurable On/Off by the configuration parameter NvMVersionInfoApi.
Hint: If source code for caller and callee of the function NvM_GetVersionInfo is available, the function should be realized as a macro. The macro should be defined in the modules header file.
The function NvM_GetVersionInfo is affected by the common block configuration parameter [NVM028]. ⌋ ( )

8.3.1.7 NvM_SetRamBlockStatus

[NVM453] ⌈

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 RAM block and mark block as changed. FALSE: Invalidate the RAM block and mark block as unchanged.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	E_OK: The status of the RAM-Block was changed as requested. E_NOT_OK: An error occurred.
Description:	Service for setting the RAM block status of an NVRAM block.	

⌋(BSW08545)

[NVM240] ⌈The function NvM_SetRamBlockStatus shall only work on NVRAM blocks with a permanently configured RAM block and shall have no effect to other NVRAM blocks.⌋(BSW08546)

[NVM241] ⌈The function NvM_SetRamBlockStatus shall assume that a changed permanent RAM block is valid (basic assumption).⌋(BSW08545)

[NVM405] ⌈When the “BlockChanged” parameter passed to the function NvM_SetRamBlockStatus is FALSE the corresponding RAM block is either invalid or unchanged (or both).⌋(BSW08545)

[NVM406] ⌈When the “BlockChanged” parameter passed to the function NvM_SetRamBlockStatus is TRUE, the corresponding permanent RAM block is valid and changed.⌋( )

[NVM121] ⌈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).⌋( )

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`.
Regarding error detection, the requirement [NVM02 is applicable to the function NvM_SetRamBlockStatus.

[NVM711] ⌈The NvM module’s environment shall have initialized the NvM module before it calls the function NvM_SetRamBlockStatus.⌋()

[NVM408] ⌈The NvM module shall provide the function `NvM_SetRamBlockStatus` only if it is configured via `NvMSetRamBlockStatusApi` [NVM028].⌋()

NVRAM common configuration parameters [NVM028], block management types [NVM061_Conf] and one configured NVRAM block descriptor are needed in the configuration with respect to the function NvM_SetRamBlockStatus [NVM062].

8.3.1.8 NvM_SetBlockLockStatus

[NVM548] ⌈

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 of an NVRAM block.	

⌋(BSW08546)

[NVM732] ⌈The function NvM_BlockLockStatus shall only work on NVRAM blocks with a permanently configured RAM block 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.⌋( )

8.3.2 Asynchronous single block requests

8.3.2.1 NvM_ReadBlock

[NVM454] ⌈

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.	

⌋(BSW08533, BSW176, BSW016)

[NVM010] ⌈The job of the function NvM_ReadBlock shall copy the data of the NV block to the corresponding RAM block.⌋(BSW016)

[NVM195] ⌈The function NvM_ReadBlock shall take over the given parameters, queue the read request in the job queue and return.⌋(BSW016)

[NVM196] ⌈If the funtion is provided with a valid RAM block address, it is used. If a NULL pointer is provided and if a permanent block or an NvMBlockUseSyncMechanism is specified the permanent block of the APIs shall be used. Otherwise a DET-Parameter error (see Section 7.4) shall be emitted.⌋(BSW016)

[NVM278] ⌈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. In this case, the parameter NvM_DstPtr must be unequal to the NULL pointer. Otherwise a DET-Parameter error (see Section 7.4) shall be emitted.⌋( )

[NVM198] ⌈The function NvM_ReadBlock shall invalidate a permanent RAM block immediately when the block is successfully enqueued or the job processing starts, i.e. copying data from NV memory or ROM to RAM.⌋( )

[NVM199] ⌈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.⌋( )

[NVM340] ⌈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.⌋( )

[NVM355] ⌈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.⌋( )

[NVM651] ⌈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.⌋( )

[NVM354] ⌈The job of the function NvM_ReadBlock shall copy the ROM block to RAM and set the job result to NVM_REQ_OK if the NVRAM block management type is NVM_BLOCK_DATASET and the dataset index points at a ROM block.⌋( )

[NVM200] ⌈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.⌋( )

[NVM366] ⌈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.⌋( )

[NVM206] ⌈The job of the function NvM_ReadBlock shall set the job result to NVM_REQ_OK if the NV block was copied successfully from NV memory to RAM.⌋( )

[NVM341] ⌈The job of the function NvM_ReadBlock shall set the request result to NVM_REQ_NV_INVALIDATED if the MemIf reports MEMIF_BLOCK_INVALID.⌋( )

[NVM652] ⌈The job of the function NvM_ReadBlock shall report no error to the DEM if the MemIf reports MEMIF_BLOCK_INVALID.⌋( )

[NVM358] ⌈The job of the function NvM_ReadBlock shall set the request result to NVM_REQ_INTEGRITY_FAILED if the MemIf reports MEMIF_BLOCK_INCONSISTENT.⌋( )

[NVM653] ⌈The job of the function NvM_ReadBlock shall report NVM_E_INTEGRITY_FAILED to the DEM if the MemIf reports MEMIF_BLOCK_INCONSISTENT.⌋( )

[NVM359] ⌈The job of the function NvM_ReadBlock shall set the request result to NVM_REQ_NOT_OK if the MemIf reports MEMIF_JOB_FAILED.⌋( )

[NVM654] ⌈The job of the function NvM_ReadBlock shall report NVM_E_REQ_FAILED to the DEM if the MemIf reports MEMIF_JOB_FAILED.⌋( )

[NVM279] ⌈The job of the function NvM_ReadBlock shall set the job 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. ⌋( )

[NVM655] ⌈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.⌋( )

[NVM316] ⌈The job of the function NvM_ReadBlock shall mark every NVRAM block that has been configured with NVM_WRITE_BLOCK_ONCE (TRUE) that is not detected by underlying SW as being invalidated, shall be marked as write protected.⌋( )

[NVM317] ⌈The job of the function NvM_ReadBlock shall invalidate a NVRAM block of management type redundant if both NV blocks have been invalidated.⌋( )

[NVM201] ⌈The job of the function NvM_ReadBlock shall request a CRC recalculation over the RAM block data after the copy process [[NVM180] if the NV block is configured with CRC, i.e. if NvMCalRamBlockCrC == TRUE for the NV block.⌋( )

[NVM202] ⌈The job of the function NvM_ReadBlock shall load the default values according to processing of NvM_RestoreBlockDefaults if the recalculated CRC is not equal to the CRC stored in NV memory.⌋( )

[NVM658] ⌈NvM_ReadBlock: If there are no default values available, the RAM blocks shall remain invalid.⌋( )

[NVM657] ⌈The job of the function NvM_ReadBlock shall load the default values according to processing of NvM_RestoreBlockDefaults if the read request passed to the underlying layer fails.⌋( )

[NVM203] ⌈The job of the function NvM_ReadBlock shall report NVM_E_INTEGRITY_FAILED to the DEM if a CRC mismatch occurs.⌋( )

[NVM204] ⌈The job of the function NvM_ReadBlock shall set the job result NVM_REQ_INTEGRITY_FAILED if a CRC mismatch occurs.⌋( )

Regarding error detection, the requirement NVM027 is applicable to the function NvM_ReadBlock.

[NVM712] ⌈The NvM module's environment shall have initialized the NvM module before it calls the function NvM_ReadBlock.⌋( )

Hint: NVRAM common block configuration parameters [NVM028], block management types [NVM061_Conf] and one configured NVRAM block descriptor are needed for configuration with respected to the function NvM_ReadBlock [NVM062].

8.3.2.2 NvM_WriteBlock

[NVM455] ⌈

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.	

⌋(BSW176, BSW017)

[NVM410] ⌈The job of the function NvM_WriteBlock shall copy the data of the RAM block to its corresponding NV block.⌋(BSW017)

[NVM411] ⌈The function NvM_WriteBlock shall test the write protection attribute of the NV block in the administrative part of the corresponding RAM block. In case of failure an NVM_E_WRITE_PROTECTED / (during production) error shall be reported.⌋(BSW017)

[NVM217] ⌈The function NvM_WriteBlock shall return with E_NOT_OK, if a write protected NVRAM block is referenced by the passed BlockId parameter.⌋( )

[NVM749] ⌈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 7.4) shall be emitted.⌋( )

[NVM208] ⌈The function NvM_WriteBlock shall take over the given parameters, queue the write request in the job queue and return.⌋(BSW08541)

[NVM209] ⌈The function NvM_WriteBlock shall check the NVRAM block protection when the request is enqueued but not again before the request is executed.⌋( )

[NVM300] ⌈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. ⌋ ( )

[NVM210] ⌈If the function is provided with a valid RAM block address, it is used. If a NULL pointer is provided and if a permanent block or an NvMBlockUseSyncMechanism is specified the permanent block of the APIs shall be used. Otherwise a DET-Parameter error (see Section 7.4) shall be emitted... ⌋ ( )

[NVM280] ⌈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. In this case, the parameter NvM_SrcPtr must be unequal to a NULL pointer. Otherwise a DET-Parameter error (see Section 7.4) shall be emitted⌋ ( )

[NVM212] ⌈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 [NVM180]. ⌋ ( )

[NVM338] ⌈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. ⌋ ( )

[NVM303] ⌈The job of the function NvM_WriteBlock shall assume a referenced permanent RAM block to be valid when the request is passed to the NvM module. If the permanent RAM block is still in an invalid state, the function NvM_WriteBlock shall validate it automatically before copying the RAM block contents to NV memory. ⌋ ( )

[NVM213] ⌈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 job result to NVM_REQ_NOT_OK. ⌋ (BSW08554)

[NVM659] ⌈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. ⌋ ( )

[NVM216] ⌈The configuration parameter NVM_MAX_NUM_OF_WRITE_RETRIES [NVM028] 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. ⌋ ( )

[NVM284] ⌈The job of the function NvM_WriteBlock shall set NVM_REQ_OK as job result if the passed BlockId references a NVRAM block of type NVM_BLOCK_REDUNDANT and one of the NV blocks have been written successfully. ⌋ ( )

[NVM328] ⌈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.
Regarding error detection, the requirement NVM027 is applicable to the function NvM_WriteBlock. ⌋ ( )

[NVM713] ⌈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.
NVRAM common block configuration parameters [NVM028], block management types [NVM061_Conf] and one configured NVRAM block descriptor are needed in the configuration with respect to the function NvM_WriteBlock [NVM062].

[NVM547] ⌈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

[NVM456] ⌈

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.	

⌋(BSW018)

[NVM012] ⌈The job of the function NvM_RestoreBlockDefaults shall restore the default data to its corresponding RAM block. ⌋(BSW018)

[NVM224] ⌈The function NvM_RestoreBlockDefaults shall take over the given parameters, queue the request in the job queue and return. ⌋( )

[NVM267] ⌈The job of the function NvM_RestoreBlockDefaults shall load the default data from a ROM block if a ROM block is configured. ⌋(BSW018)

[NVM266] ⌈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. ⌋(BSW018)

[NVM353] ⌈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. ⌋( )

[NVM435] ⌈If the function is provided with a valid RAM block address, it is used. If a NULL pointer is provided and if a permanent block or an NvMBlockUseSyncMechanism is specified the permanent block of the APIs shall be used. Otherwise a DET-Parameter error (see Section 7.4) shall be emitted. ⌋( )

[NVM436] ⌈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. Otherwise a DET-Parameter error (see Section 7.4) shall be emitted⌋ ( )

[NVM227] ⌈The job of the function NvM_RestoreBlockDefaults shall invalidate a RAM block before copying default data to the RAM if a permanent RAM block is requested. ⌋ ( )

[NVM228] ⌈The job of the function NvM_RestoreBlockDefaults shall validate and assume a RAM block to be changed if the requested RAM block is permanent and the copy process of the default data to RAM was successful. ⌋ ( )

[NVM229] ⌈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. ⌋ ( )
Regarding error detection, the requirement NVM027 is applicable to the function NvM_RestoreBlockDefaults.

[NVM714] ⌈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).
NVRAM common block configuration parameters [NVM028], block management types [NVM061_Conf] and one configured NVRAM block descriptor are needed in the configuration with respect to the function NvM_RestoreBlockDefaults [NVM062].

8.3.2.4 NvM_EraseNvBlock

[NVM457] ⌈

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.	

⌋(BSW08544)

[NVM415] ⌈The job of the function NvM_EraseNvBlock shall erase a NV block. ⌋(BSW08544)

[NVM231] ⌈The function NvM_EraseNvBlock shall take over the given parameters, queue the request and return. ⌋( )

[NVM418] ⌈The function NvM_EraseNvBlock shall queue the request to erase in case of disabled write protection. ⌋( )

[NVM416] ⌈The job of the function NvM_EraseNvBlock shall leave the content of the RAM block unmodified. ⌋( )

[NVM417] ⌈The function NvM_EraseNvBlock shall test the write protection attribute of the NV block in the corresponding Administrative block. In case of failure an NVM_E_WRITE_PROTECTED / (during production) error shall be reported. ⌋( )

[NVM262] ⌈The function NvM_EraseNvBlock shall return with E_NOT_OK if a write protected NV block is referenced. ⌋( )

[NVM661] ⌈The function NvM_EraseNvBlock shall return with E_NOT_OK if a ROM block of a dataset NVRAM block is referenced. ⌋( )

[NVM230] ⌈The function NvM_EraseNvBlock shall check the write protection attribute of a NVRAM block only before the job is put to the job queue. In case of

failure an NVM_E_WRITE_PROTECTED / (during production) error shall be reported.」( )

[NVM662] 「NvM_EraseNvBlock: The NvM module shall not re-check the write protection before fetching the job from the job queue.」( )

[NVM269] 「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.」( )

[NVM271] 「The job of the function NvM_EraseNvBlock shall set the job result to NVM_REQ_NOT_OK if the processing of the service fails.」( )

[NVM663] 「The job of the function NvM_EraseNvBlock shall report NVM_E_REQ_FAILED to the DEM if the processing of the service fails.」( )

[NVM357] 「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.」( )

Regarding error detection, the requirement NVM027 is applicable to the function NvM_EraseNvBlock.

[NVM715] 「The NvM module's environment shall have initialized the NvM module before it calls the function NvM_EraseNvBlock.」( )
NVRAM common block configuration parameters [NVM028], block management types [NVM061_Conf] and one configured NVRAM block descriptor are needed in the configuration with respect to the function NvM_EraseNvBlock [NVM062].

8.3.2.5 NvM_CancelWriteAll

[NVM458] ⌈

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.

⌋(BSW08558, BSW08540)

[NVM019] ⌈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⌋ (BSW08540)

[NVM232] ⌈The function NvM_CancelWriteAll shall signal the request to the NvM module and return. ⌋( )

[NVM233] ⌈The function NvM_CancelWriteAll shall be without any effect if no NvM_WriteAll request is pending. ⌋( )

[NVM234] ⌈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. ⌋( )

[NVM235] ⌈The request result of the function NvM_CancelWriteAll shall be implicitly given by the result of the NvM_WriteAll request to be canceled. ⌋( )

[NVM255] ⌈The function NvM_CancelWriteAll shall ignore an already pending NvM_CancelWriteAll request. ⌋( )

[NVM236] ⌈NvM_CancelWriteAll shall only modify the error/status attribute field of the pending and currently written blocks. ⌋( )

Regarding error detection, the requirement NVM027 is applicable to the function NvM_CancelWriteAll.

[NVM716] ⌈The NvM module's environment shall have initialized the NvM module before it calls the function function NvM_CancelWriteAll.⌋( )

[NVM420] ⌈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.⌋( )

8.3.2.6 NvM_InvalidateNvBlock

[NVM459] ⌈

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.	

⌋(BSW08011)

[NVM421] ⌈The job of the function NvM_InvalidateNvBlock shall invalidate a NV block.⌋(BSW08011)

[NVM422] ⌈The job of the function NvM_InvalidateNvBlock shall leave the RAM block unmodified.⌋( )

[NVM423] ⌈The function NvM_InvalidateNvBlock shall check the write protection attribute of the NV block in the administrative part of the corresponding RAM block. In case of failure an NVM_E_WRITE_PROTECTED / (during production) error shall be reported.⌋( )

[NVM424] ⌈The function NvM_InvalidateNvBlock shall queue the request if the write protection of the corresponding NV block is disabled.⌋( )

[NVM239] ⌈The function NvM_InvalidateNvBlock shall take over the given parameters, queue the request and return.⌋( )

[NVM272] ⌈The function NvM_InvalidateNvBlock shall return with E_NOT_OK if a write protected NV block is referenced by the BlockId parameter.⌋( )

[NVM664] ⌈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.⌋( )

[NVM273] ⌈The function NvM_EraseNvBlock shall check the write protection attribute of a NVRAM block only before the job is put to the job queue. In case of failure an NVM_E_WRITE_PROTECTED / (during production) error shall be reported.s ⌋( )

[NVM665] ⌈The NvM module shall not recheck write protection before fetching the job from the job queue. ⌋( )

[NVM274] ⌈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. ⌋( )

[NVM275] ⌈The function NvM_InvalidateNvBlock shall set the job result to NVM_REQ_NOT_OK if the processing of this service fails. ⌋( )

[NVM666] ⌈The function NvM_InvalidateNvBlock shall report NVM_E_REQ_FAILED to the DEM if the processing of this service fails. ⌋( )

Regarding error detection, the requirement NVM027 is applicable to the function NvM_InvalidateNvBlock.

[NVM717] ⌈The NvM module's environment shall have initialized the NvM module before it calls the function function NvM_InvalidateNvBlock. ⌋( )

NVRAM common block configuration parameters [NVM028], block management types [NVM061_Conf] and one configured NVRAM block descriptor are needed in the configuration with respect to the function NvM_InvalidateBlock [NVM062].

8.3.2.7 NvM_CancelJobs

[NVM535] ⌈

Service name:	NvM_CancelJobs	
Syntax:	Std_ReturnType NvM_CancelJobs(NvM_BlockIdType BlockId)	
Service ID[hex]:	0x10	
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 cancel all jobs pending for a NV block.	

⌋(BSW08560, BSW08559)

[NVM536] ⌈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. ⌋(BSW08560, BSW08559)

[NVM537] ⌈A currently processed job shall continue even after the call of NvM_CancelJobs. ⌋( )

[NVM225] ⌈ The job of the function NvM_CancelJobs shall set block specific job result for specified NVRAM block to NVM_REQ_CANCELED in advance 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. ⌋( )

8.3.3 Asynchronous multi block requests

8.3.3.1 NvM_ReadAll

[NVM460] ⌈

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.

⌋(BSW08533, BSW176)

[NVM356] ⌈The multi block service NvM_ReadAll shall provide two distinct functionalities.

- Initialize the management data for all NVRAM blocks (see NVM304 ff)
- Copy data to the permanent RAM blocks for those NVRAM blocks which are configured accordingly.

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

[NVM243] ⌈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. ⌋( )

[NVM304] ⌈The job of the function NvM_ReadAll shall set each proceeding block specific job result for NVRAM blocks in advance. ⌋( )

[NVM667] ⌈The job of the function NvM_ReadAll shall set the multi block job result to NVM_REQ_PENDING in advance. ⌋( )

[NVM244] ⌈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. ⌋( )

[NVM245] ⌈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. ⌋( )

[NVM362] ⌈The NvM module shall initiate the recalculation of the RAM CRC for every NVRAM block with a valid permanent RAM block and NvMCalcRamBlockCrc == TRUE during the processing of NvM_ReadAll. ⌋( )

[NVM364] ⌈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. ⌋( )

[NVM363] ⌈The job of the function NvM_ReadAll shall load default values in case the blocks are marked as invalid or if the recalculated CRC does not match. ⌋( )

[NVM246] ⌈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. ⌋( )

[NVM669] ⌈NvM_ReadAll: The NVRAM block with the block ID 1 (redundant type with CRC) shall be reserved to contain the stored NVRAM configuration ID. ⌋( )

[NVM247] ⌈The job of the function NvM_ReadAll shall process the normal runtime preparation for all configured NVRAM blocks in case of configuration ID match. ⌋( )

[NVM670] ⌈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. ⌋( )

[NVM305] ⌈The job of the function NvM_ReadAll shall report the 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. ⌋( )

[NVM671] ⌈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. ⌋( )

[NVM307] ⌈The job of the function NvM_ReadAll shall set the error/status information field of a NVRAM block's administrative block to NVM_REQ_NOT_OK in the case of configuration ID mismatch. ⌋( )

[NVM306] ⌈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 a production error to the DEM. ⌋( )

[NVM672] ⌈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. ⌋( )

[NVM673] ⌈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. ⌋( )

[NVM248] ⌈The job of the function NvM_ReadAll shall ignore a configuration ID mismatch and behave normal if NvMDynamicConfiguration == FALSE [NVM028]. ⌋( )

[NVM249] ⌈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. ⌋( )

[NVM674] ⌈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. ⌋( )

[NVM314] ⌈The job of the function NvM_ReadAll shall mark every NVRAM block that has been configured with NVM_WRITE_BLOCK_ONCE (TRUE) and that is not detected by underlying SW as being invalidated, shall be marked as write protected. This write protection cannot be cleared by NvM_SetBlockProtection. ⌋( )

[NVM315] ⌈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. ⌋( )

[NVM718] ⌈ The NvM module's environment shall use the multi block request NvM_ReadAll to load and validate the content of configured permanent RAM blocks during start-up [NVM091]. ⌋( )

[NVM118] ⌈The job of the function NvM_ReadAll shall process only the permanent RAM blocks which are configured with NvMSelectBlockForReadAll == TRUE. ⌋( )

[NVM287] ⌈The job of the function NvM_ReadAll shall set the job result to NVM_REQ_BLOCK_SKIPPED for all NVRAM blocks which are not loaded automatically during processing of the NvM_ReadAll job.⌋( )

[NVM426] ⌈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.⌋( )

[NVM427] ⌈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.⌋( )

[NVM308] ⌈The job of the function NvM_ReadAll shall load the ROM default data to the corresponding RAM blocks and set the error/status field in the administrative block to NVM_REQ_OK when processing the extended runtime preparation.⌋( )

[NVM309] ⌈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.⌋( )

[NVM310] ⌈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 NvM_WriteAll and request a CRC recalculation if a configuration ID mismatch occurs and if the NVRAM block is configured with NvMDynamicConfiguration == TRUE.⌋( )

[NVM311] ⌈The NvM module shall allow applications to send any request for the reserved NVRAM Block ID 1, including NvM_WriteBlock.⌋( )

[NVM312] ⌈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.⌋( )

[NVM313] ⌈In case of a Configuration ID match, the job of the function NvM_ReadAll shall not automatically write to the Configuration ID block stored in the reserved NVRAM block 1.⌋( )

[NVM288] ⌈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 [NVM118], where the read attempt of the first block fails (see also [NVM531]).⌋( )

[NVM290] ⌈The job of the function NvM_ReadAll shall set the block specific job result to NVM_REQ_OK if the job has successfully copied the corresponding NV block from NV memory to RAM. ⌋( )

[NVM342] ⌈The job of the function NvM_ReadAll shall set the block specific job result to NVM_REQ_NV_INVALIDATED if the MemIf reports MEMIF_BLOCK_INVALID. ⌋( )

[NVM676] ⌈The job of the function NvM_ReadAll shall report no error to the DEM if the MemIf reports MEMIF_BLOCK_INVALID. ⌋( )

[NVM360] ⌈The job of the function NvM_ReadAll shall set the block specific job result to NVM_REQ_INTEGRITY_FAILED if the MemIf reports MEMIF_BLOCK_INCONSISTENT. ⌋( )

[NVM677] ⌈The job of the function NvM_ReadAll shall report NVM_E_INTEGRITY_FAILED to the DEM if the MemIf reports MEMIF_BLOCK_INCONSISTENT. ⌋( )

[NVM361] ⌈The job of the function NvM_ReadAll shall set the block specific job result to NVM_REQ_NOT_OK if the MemIf reports MEMIF_JOB_FAILED. ⌋( )

[NVM678] ⌈The job of the function NvM_ReadAll shall report NVM_E_REQ_FAILED to the DEM, if the MemIf reports MEMIF_JOB_FAILED. ⌋( )

[NVM291] ⌈The job of the function NvM_ReadAll shall set the block specific job 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. ⌋( )

[NVM292] ⌈The job of the function NvM_ReadAll shall request a CRC recalculation over the RAM block data after the copy process [NVM180] if the NV block is configured with CRC, , i.e. if NvMCalRamBlockCrC == TRUE for the NV block. ⌋( )

[NVM293] ⌈The job of the function NvM_ReadAll shall load the default values to the RAM blocks according to the processing of NvM_RestoreBlockDefaults if the recalculated CRC is not equal to the CRC stored in NV memory and if the default values are available. ⌋( )

[NVM679] ⌈The job of the function NvM_ReadAll shall load the default values to the RAM blocks according to the processing of NvM_RestoreBlockDefaults if the read

request passed to the underlying layer fails and if the default values are available.⌋( )

[NVM680] ⌈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.⌋( )

[NVM294] ⌈The job of the function NvM_ReadAll shall report NVM_E_INTEGRITY_FAILED to the DEM if a CRC mismatch occurs.⌋( )

[NVM295] ⌈The job of the function NvM_ReadAll shall set a block specific job result to NVM_REQ_INTEGRITY_FAILED if a CRC mismatch occurs.⌋( )

[NVM302] ⌈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.⌋( )

[NVM301] ⌈The job of the function NvM_ReadAll shall set the multi block job result to NVM_REQ_NOT_OK if the job fails with the processing of at least one NVRAM block.⌋( )

[NVM281] ⌈If configured by NvMSingleBlockCallback, the job of the function NvM_ReadAll shall call the single block callback after having completely processed a NVRAM block.⌋( )

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

- A single-block callback issued from a multi-block request (e.g. NvM_ReadAll) will result in an RTE event.
- If the RTE is initialized after or during the asynchronous multi-block request (e.g. NvM_ReadAll), all or some of these RTE events will get lost because they are overwritten during the RTE initialization (see rte_sws_2536).
- After its initialization, the RTE can use the "surviving" RTE events to start software components even before the complete multi-block request (e.g. 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 multi-block requests.

[NVM251] ⌈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.⌋( )

[NVM367] ⌈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.⌋( )

[NVM719] ⌈The NvM module's environment shall have initialized the NvM module before it calls the function NvM_ReadAll.⌋( )

Regarding error detection, the requirement NVM027 is applicable to the function NvM_ReadAll.
The DEM shall already be able to accept error notifications.
NVRAM common block configuration parameters [NVM028], block management types [NVM061_Conf] and all configured NVRAM block descriptors are needed in the configuration with respect to the function NvM_ReadAll [NVM062], [NVM069].

8.3.3.2 NvM_WriteAll

[NVM461] ⌈

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.

⌋(BSW176, BSW08535)

[NVM018] ⌈The job of the function NvM_WriteAll shall synchronize the contents of permanent RAM blocks to their corresponding NV blocks on shutdown. ⌋ (BSW08535)

[NVM733] ⌈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. ⌋( )

[NVM254] ⌈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. ⌋( )

[NVM549] ⌈The job of the function NvM_ WriteAll shall set each proceeding block specific job result for NVRAM blocks and the multi block job result to NVM_REQ_PENDING in advance. ⌋( )

[NVM252] ⌈The job of the function NvM_WriteAll shall process only the permanent RAM blocks for which the corresponding NVRAM block parameter NvMSelectBlockForWriteAll is configured to true. ⌋( )

[NVM430] ⌈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. ⌋( )

[NVM431] ⌈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. ⌋( )

[NVM681] ⌈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.⌋( )

[NVM432] ⌈The job of the function NvM_WriteAll shall check the write-protection for each RAM block in advance.⌋( )

[NVM682] ⌈The job of the function NvM_WriteAll shall check the “valid/modified” state for each RAM block in advance.⌋( )

[NVM433] ⌈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.⌋( )

[NVM474] ⌈The job of the function NvM_WriteAll shall correct the redundant data (if configured) if the redundancy has been lost.⌋( )

[NVM434] ⌈The job of the function NvM_WriteAll shall skip every write-protected NVRAM block without error notification.⌋( )

[NVM750] ⌈ The job of the function NvM_WriteAll shall skip every locked NVRAM block without error notification.⌋( )

[NVM298] ⌈The job of the function NvM_WriteAll shall set the job result for each NVRAM block which has not been written automatically by the job to NVM_REQ_BLOCK_SKIPPED.⌋( )

[NVM339] ⌈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.⌋( )

[NVM253] ⌈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.⌋(BSW08535)

[NVM296] ⌈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.⌋( )

[NVM683] ⌈The job of the function NvM_WriteAll shall set the block specific job result to NVM_REQ_NOT_OK if the write retry counter becomes greater than the configured NVM_MAX_NUM_OF_WRITE_RETRIES.⌋( )

[NVM684] ⌈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.⌋( )

[NVM337] ⌈The job of the function NvM_WriteAll shall set the single block job result to NVM_REQ_OK if the processed NVRAM block is of type NVM_BLOCK_REDUNDANT and one of the NV blocks has been written successfully.⌋( )

[NVM238] ⌈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.⌋( )

[NVM237] ⌈The NvM module shall set the multi block request result to NVM_REQ_CANCELED in case of cancellation of NvM_WriteAll.⌋( )

[NVM685] ⌈NvM_WriteAll: The NvM module shall anyway report the error code condition, due to a failed NVRAM block write, to the DEM.⌋( )

[NVM318] ⌈The job of the function NvM_WriteAll shall set the multi block request result to NVM_REQ_NOT_OK if processing of one or even more NVRAM blocks fails.⌋( )

[NVM329] ⌈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.⌋( )
Regarding error detection, the requirement NVM027 is applicable to the function NvM_WriteAll.

[NVM720] ⌈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.

NVRAM common block configuration parameters [NVM028], block management types [NVM061_Conf] and all configured NVRAM block descriptors are needed in the configuration with respect to the NvM_WriteAll function [NVM062], [NVM069].

8.4 Call-back notifications

8.4.1 Callback notification of the NvM module

[NVM438] ⌈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.

Note: The file NvM_Cbk.h is to be included by the underlying memory driver layers. ⌋ ( )

8.4.1.1 NVRAM Manager job end notification without error

[NVM462] ⌈

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.

⌋ ( )

[NVM111] ⌈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). ⌋ ( )

[NVM440] ⌈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 [NVM028] configuration parameters. ⌋ ( )

8.4.1.2 NVRAM Manager job end notification with error

[NVM463] ⌈

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.

⌋(BSW125)

[NVM112] ⌈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).⌋( )

[NVM441] ⌈The NvM module shall only provide the callback function NvM_JobErrorNotification if polling mode is disabled via NvMPollingMode.
The function NvM_JoberrorNotification is affected by the common [NVM028] configuration parameters.⌋( )

8.5 Scheduled functions

These functions are directly called by the Basic Software Scheduler. The following functions shall have no return value and no parameter. All functions shall be non reentrant.

[NVM322] ⌈BSW module main processing functions are only allowed to be allocated to basic tasks.⌋(BSW00424)

[NVM464] ⌈

Service name:	NvM_MainFunction
Syntax:	void NvM_MainFunction(void)
Service ID[hex]:	0x0e
Timing:	VARIABLE_CYCLIC
Description:	Service for performing the processing of the NvM jobs.

⌋(BSW00425, BSW00373, BSW00376, BSW172)

[NVM256] ⌈The function NvM_MainFunction shall perform the processing of the NvM module jobs.⌋( )

[NVM333] ⌈The function NvM_MainFunction shall perform the CRC recalculation if requested for a NVRAM block in addition to NVM256.⌋( )

[NVM334] ⌈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.⌋( )

[NVM257] ⌈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.⌋( )

[NVM258] ⌈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 [NVM182].⌋( )

[NVM259] ⌈The function NvM_MainFunction shall supervise the immediate priority queue (if configured) regarding the existence of immediate priority requests.⌋( )

[NVM346] ⌈If polling mode is enabled, the function NvM_MainFunction shall check the status of the requested job sent to the lower layer.⌋( )

[NVM347] ⌈If callback routines are configured, the function NvM_MainFunction shall call callback routines to the upper layer after completion of an asynchronous service. ⌋( )

[NVM350] ⌈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 [7]. 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. ⌋( )

[NVM349] ⌈The function NvM_MainFunction shall return immediately if no further job processing is possible. ⌋( )

[NVM721] ⌈NVRAM blocks with immediate priority are not expected to be configured to have a CRC. ⌋( )

[NVM324] ⌈The NvM module's environment does not have to execute the function NvM_MainFunction in a specific order or sequence with respect to other BSW main processing function(s). ⌋(BSW00428, BSW00433, BSW172)
The function NvM_MainFunction is affected by the common [NVM028] configuration parameters.
Terms and definitions:

Fixed cyclic: Fixed cyclic means that one cycle time is defined at configuration and shall not be changed because functionality is requiring that fixed timing (e.g. filters).
Variable cyclic: Variable cyclic means that the cycle times are defined at configuration but might be mode dependent and therefore vary during runtime.
On pre-condition: On pre-condition means that no cycle time can be defined. The function is called when the conditions are fulfilled. Alternatively, the function may be called cyclically. However, the cycle time is assigned dynamically during runtime by other modules.

8.6 Expected Interfaces

In this chapter, all interfaces required by other modules are listed.

8.6.1 Mandatory Interfaces

The following table defines all interfaces which are required to fulfill the core functionality of the module.

[NVM465] ⌈

API function	*Description*
EcuM_CB_NfyNvMJobEnd	Used to notify about the end of NVRAM jobs initiated by EcuM The callback must be callable from normal and interrupt execution contexts.
MemIf_Cancel	Invokes the "Cancel" function of the underlying memory abstraction module selected by the parameter DeviceIndex.
MemIf_EraseImmediateBlock	Invokes the "EraseImmediateBlock" function of the underlying memory abstraction module selected by the parameter DeviceIndex.
MemIf_GetJobResult	Invokes the "GetJobResult" function of the underlying memory abstraction module selected by the parameter DeviceIndex.
MemIf_GetStatus	Invokes the "GetStatus" function of the underlying memory abstraction module selected by the parameter DeviceIndex.
MemIf_InvalidateBlock	Invokes the "InvalidateBlock" function of the underlying memory abstraction module selected by the parameter DeviceIndex.
MemIf_Read	Invokes the "Read" function of the underlying memory abstraction module selected by the parameter DeviceIndex.
MemIf_Write	Invokes the "Write" function of the underlying memory abstraction module selected by the parameter DeviceIndex.

⌋ (BSW00383, BSW00384, BSW00421)

8.6.2 Optional Interfaces

The following table defines all interfaces which are required to fulfill an optional functionality of the module.

[NVM466] ⌈

API function	*Description*
Crc_CalculateCRC16	This service makes a CRC16 calculation on Crc_Length data bytes.
Crc_CalculateCRC32	This service makes a CRC32 calculation on Crc_Length data bytes.
Crc_CalculateCRC8	This service makes a CRC8 calculation on Crc_Length data bytes, with SAE J1850 parameters
Dem_ReportErrorStatus	Queues the reported events from the BSW modules (API is only used by BSW modules). The interface has an asynchronous behavior, because the processing of the event is done within the Dem main function.
Det_ReportError	Service to report development errors.
MemIf_SetMode	Invokes the "SetMode" functions of all underlying memory abstraction modules.

⌋ (BSW00383, BSW00384)

8.6.3 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.

[NVM113] ⌈The notification of a caller via an asynchronous callback routine (NvMSingleBlockCallback) shall be optionally configurable for all NV blocks (see NVM061_Conf). ⌋( )

[NVM740] ⌈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. ⌋( )

[NVM741] ⌈The ID identifying the NVRAM service, shall be passed to the callback routine. ⌋( )

[NVM742] ⌈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. ⌋( )

[NVM260] ⌈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). ⌋( )

[NVM686] ⌈The ID identifying the NVRAM service shall be passed to the common callback routine (NvMMultiBlockCallback). ⌋( )

8.6.3.1 Single block job end notification

[NVM467] ⌈

Service name:	NvM_SingleBlockCallbackFunction	
Syntax:	Std_ReturnType NvM_SingleBlockCallbackFunction(uint8 ServiceId, NvM_RequestResultType JobResult)	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	ServiceId	Unique Service ID of NVRAM manager service.
	JobResult	Covers the job result of the previous processed single block job.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	E_OK: Callback function has been processed successfully. E_NOT_OK: Callback function has not been processed successfully.
Description:	Per block callback routine to notify the upper layer that an asynchronous single block request has been finished.	

⌋( )

[NVM368] ⌈The single block callback function shall always return with E_OK.
There is no need for the NvM module to evaluate the return value of the single block callback function because of NVM368. ⌋( )

[NVM330] ⌈The single block callback function shall be a function pointer.
Note: Please refer to NvMSingleBlockCallback in chapter 10. The Single block job end notification might be called in interrupt context, depending on the calling function. ⌋(BSW00387)

8.6.3.2 Multi block job end notification

[NVM468] ⌈

Service name:	NvM_MultiBlockCallbackFunction	
Syntax:	void NvM_MultiBlockCallbackFunction(uint8 ServiceId, NvM_RequestResultType JobResult)	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	ServiceId	Unique Service ID of NVRAM manager service.
	JobResult	Covers the job result of the previous processed multi block job.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	None	
Description:	Common callback routine to notify the upper layer that an asynchronous multi block request has been finished.	

⌋( )

[NVM331] ⌈The Multi block job end notification shall be a function pointer.
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. ⌋(BSW00387)

8.6.3.3 Callback function for block initialization

[NVM469] ⌈

Service name:	InitBlockCallbackFunction	
Syntax:	Std_ReturnType InitBlockCallbackFunction(void)	
Sync/Async:	Synchronous	
Reentrancy:	Non Reentrant	
Parameters (in):	None	
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	E_OK: callback function has been processed successfully E_NOT_OK: callback function has not been processed successfully
Description:	Per block callback routine which shall be called by the NvM module to copy default data to a RAM block if a ROM block isn't configured.	

⌋ ( )

[NVM369] ⌈The Init block callback for block initialization shall always return with E_OK.
There is no need for the NvM module to evaluate the return value of the callback function because of NVM369. ⌋ ( )

[NVM352] ⌈The Init block callback shall be a function pointer.
Note: Please refer to NvMInitBlockCallback in chapter 10. The init block callback function might be called in interrupt context. ⌋ ( )

8.6.3.4 Callback function for RAM to NvM copy

[NVM539] ⌈

Service name:	NvM_WriteRamBlockToNvm	
Syntax:	Std_ReturnType NvM_WriteRamBlockToNvm(void* NvMBuffer)	
Service ID[hex]:	--	
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 E_NOT_OK: callback function has not been processed successfully
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.	

⌋ ( )

[NVM541] ⌈The RAM to NvM copy callback shall be a function pointer.⌋( )
Note: Please refer to NvMWriteRamBlockToNvM in chapter 10.

8.6.3.5 Callback function for NvM to RAM copy

[NVM540] ⌈

Service name:	NvM_ReadRamBlockFromNvm	
Syntax:	Std_ReturnType NvM_ReadRamBlockFromNvm(const void* NvMBuffer)	
Service ID[hex]:	--	
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 E_NOT_OK: callback function has not been processed successfully
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.	

⌋(BSW08533, BSW176)

[NVM542] ⌈The NvM to RAM copy callback shall be a function pointer.⌋( )
Note: Please refer to NvMReadRamBlockFromNvM in chapter 10.

8.7 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(…)	- 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(...)	- 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

9 Sequence Diagrams

9.1 Synchronous calls

9.1.1 NvM_Init

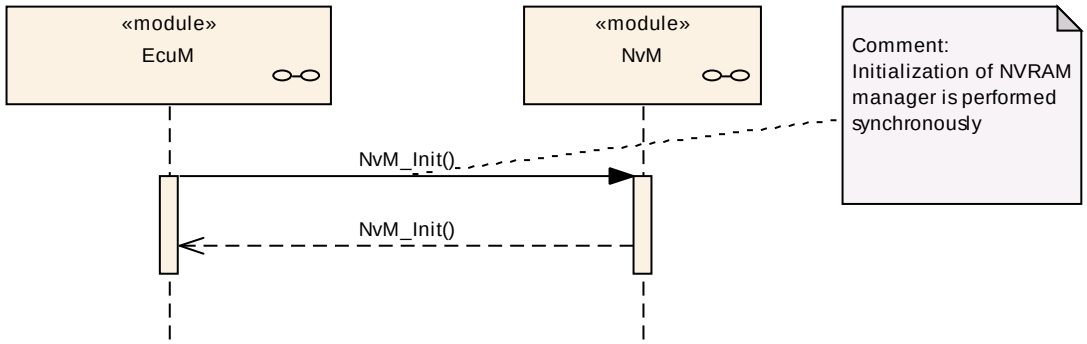

Figure 12: UML sequence diagram NvM_Init

9.1.2 NvM_SetDataIndex

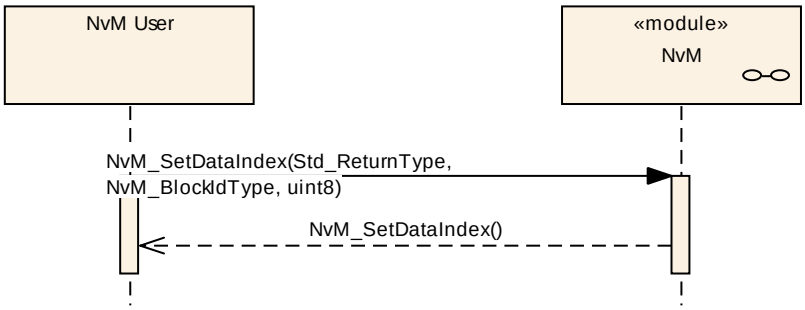

Figure 13: UML sequence diagram NvM_SetDataIndex

9.1.3 NvM_GetDataIndex

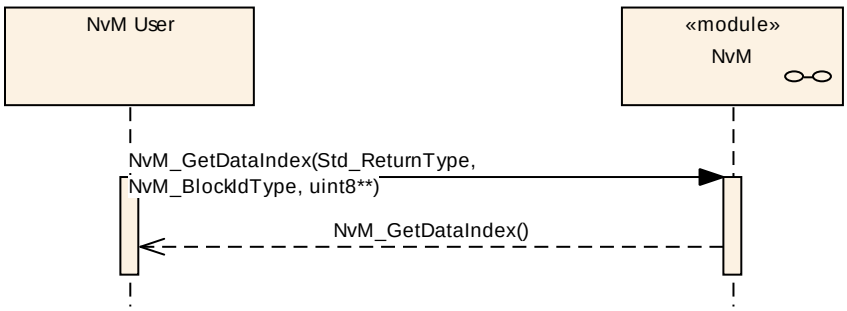

Figure 14: UML sequence diagram NvM_GetDataIndex

9.1.4 NvM_SetBlockProtection

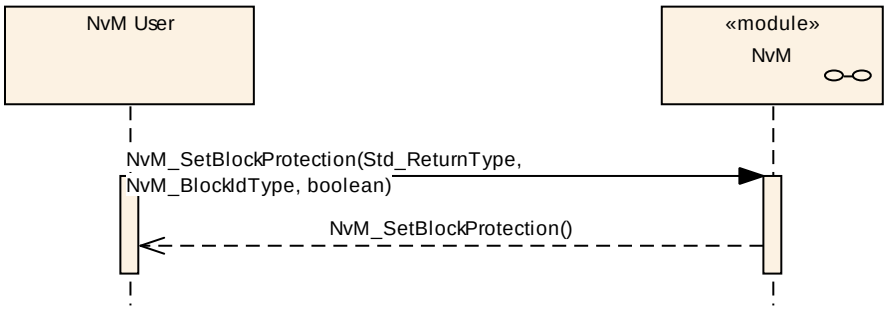

Figure 15: UML sequence diagram NvM_SetBlockProtection

9.1.5 NvM_GetErrorStatus

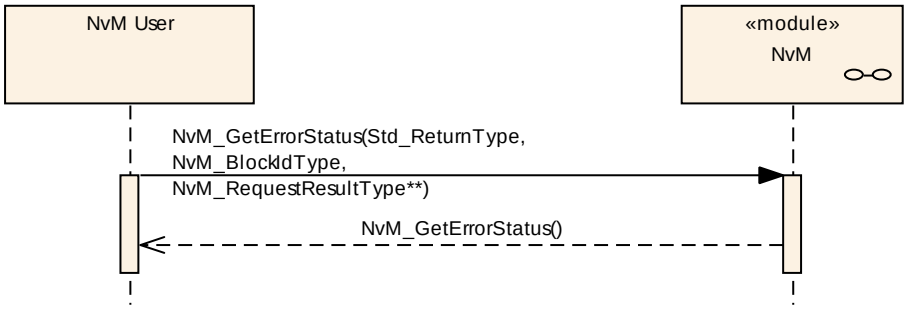

Figure 16: UML sequence diagram NvM_GetErrorStatus

9.1.6 NvM_GetVersionInfo

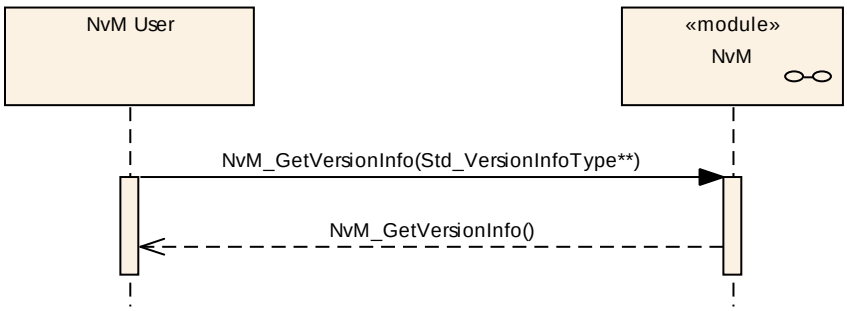

Figure 17: UML sequence diagram NvM_GetVersionInfo

9.2 Asynchronous calls

The following sequence diagrams concentrate on the interaction between the NvM module and SW-C's or the ECU state manager. For interaction regarding the Memory Interface please ref. to [5] or [6].

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.

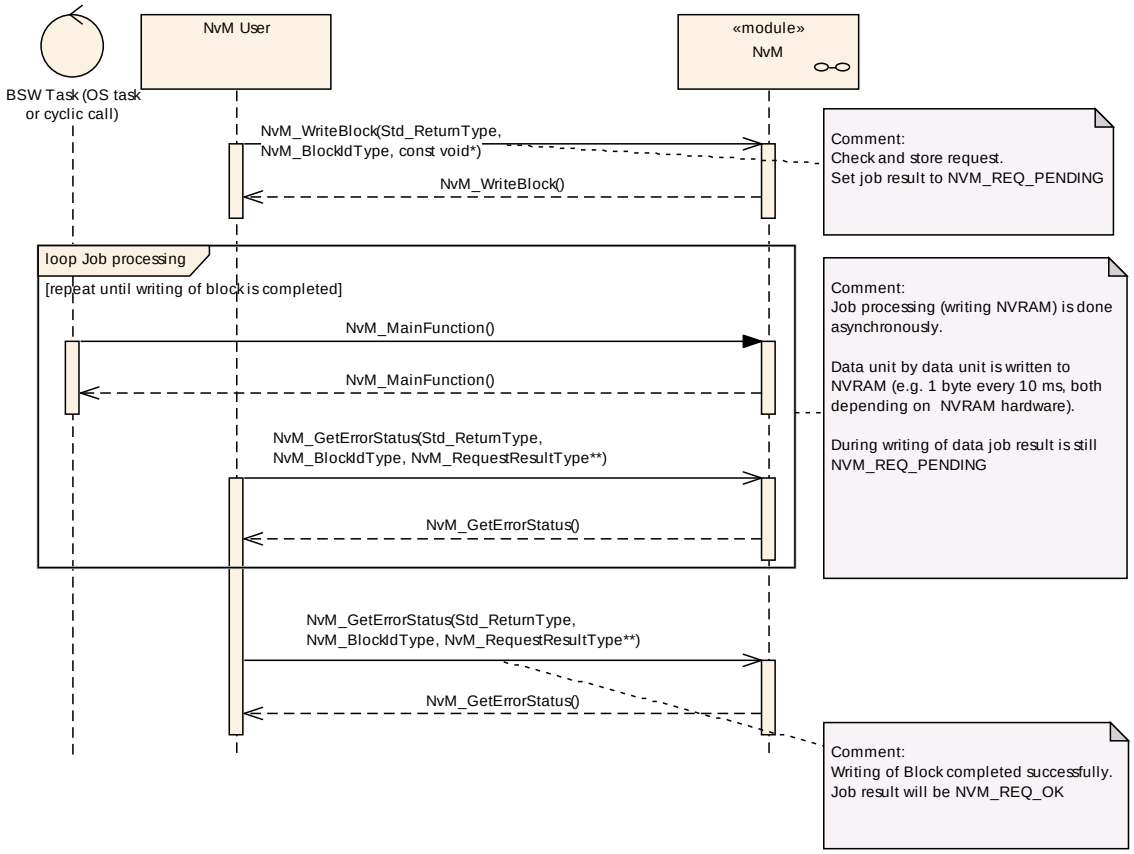

Figure 18: UML sequence diagram for asynchronous call with polling

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 (callback) by requesting the error/status information.

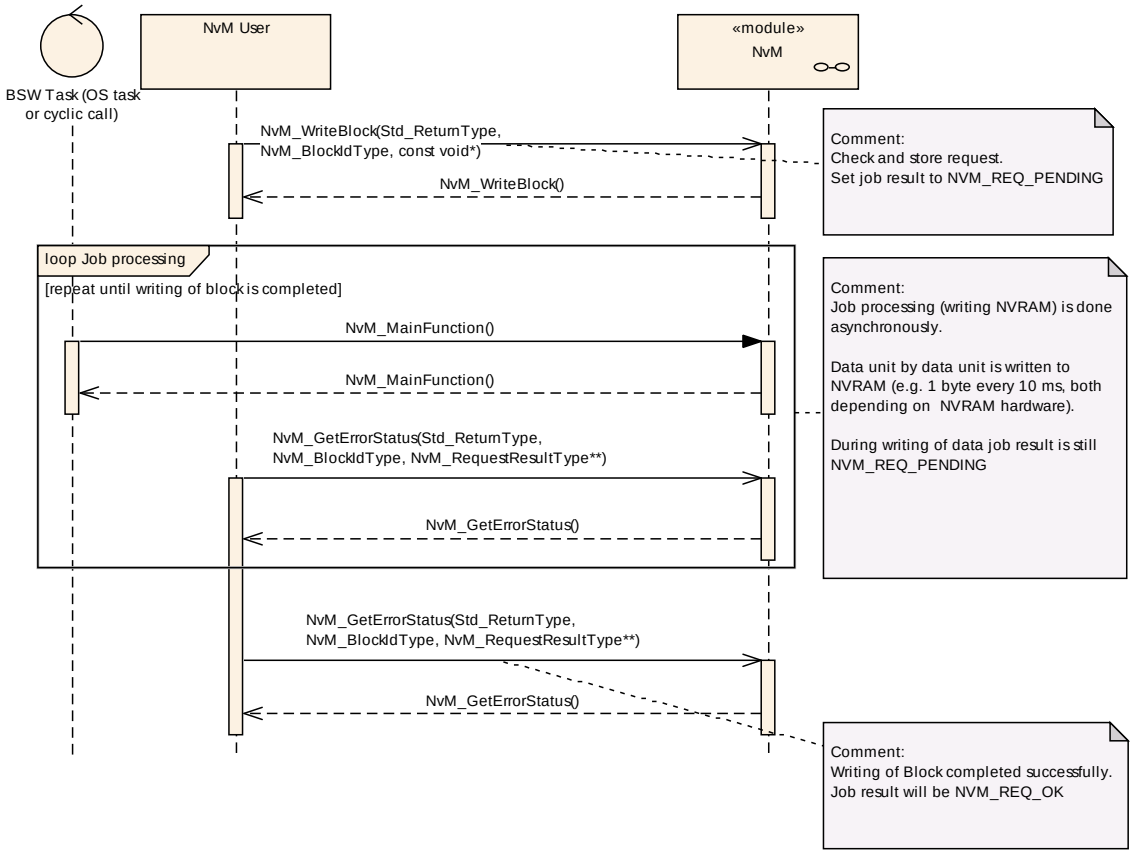

Figure 19: UML sequence diagram for asynchronous call with callback

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.

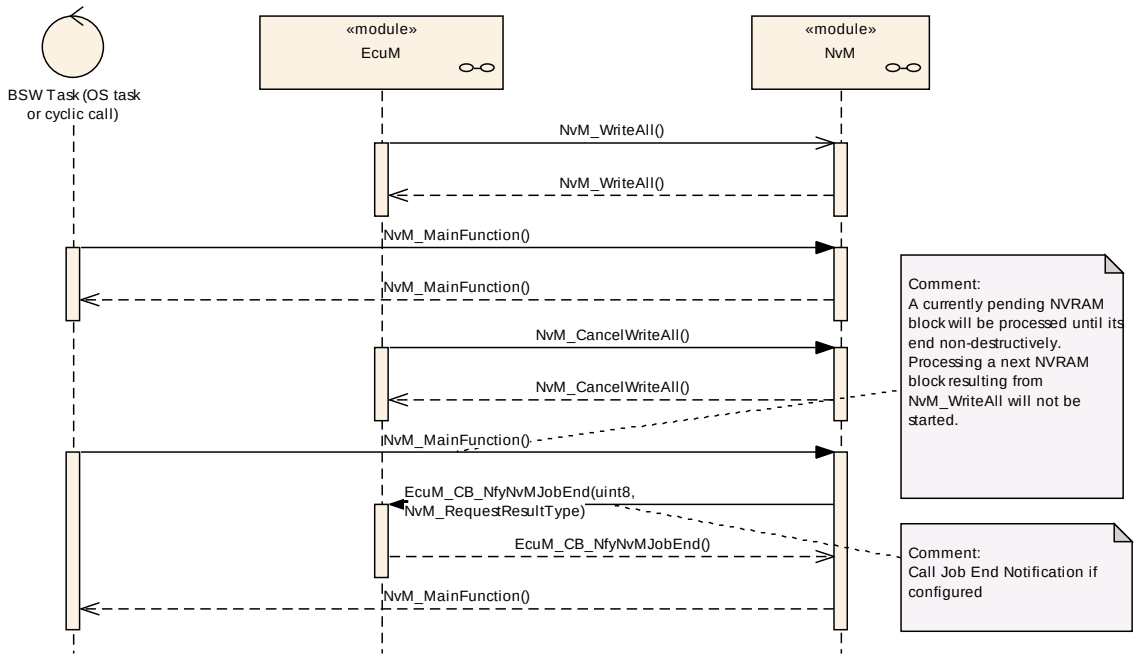

Figure 20: UML sequence diagram for cancellation of asynchronous call

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.

Chapter 10.2 specifies the structure (containers) and the parameters of the module NvM.
Chapter 10.2.8 specifies published information of the module NvM.

10.1 How to read this chapter

In addition to this section, it is highly recommended to read the documents:

- AUTOSAR Layered Software Architecture [1]
- AUTOSAR ECU Configuration Specification [10]
  This document describes the AUTOSAR configuration methodology and the AUTOSAR configuration metamodel in detail.

The following is only a short survey of the topic and it will not replace the ECU Configuration Specification document.

10.1.1 Configuration and configuration parameters

Configuration parameters define the variability of the generic part(s) of an implementation of a module. This means that only generic or configurable module implementation can be adapted to the environment (software/hardware) in use during system and/or ECU configuration.
The configuration of parameters can be achieved at different times during the software process: before compile time, before link time or after build time. In the following, the term “configuration class” (of a parameter) shall be used in order to refer to a specific configuration point in time.

10.1.2 Variants

Variants describe sets of configuration parameters. E.g., variant 1: only pre-compile time configuration parameters; variant 2: mix of pre-compile- and post build time-configuration parameters. In one variant a parameter can only be of one configuration class.

10.1.3 Containers

Containers structure the set of configuration parameters. This means:

- all configuration parameters are kept in containers.
- (sub-) containers can reference (sub-) containers. It is possible to assign a multiplicity to these references. The multiplicity then defines the possible number of instances of the contained parameters.

10.2 Containers and configuration parameters

The following chapters summarize all configuration parameters. The detailed meanings of the parameters describe chapter 7.2 and chapter 8.

10.2.1 Variants

[NVM727] ⌈The NVM module shall support the configuration variants VARIANT-PRE-COMPILE and VARIANT-LINK-TIME.
The VARIANT-PRE-COMPILE is designed for all parameters to be fixed at compile time.
The VARIANT-LINK-TIME is designed for the use cases where parameters are fixed at link-time. This variant is particularly useful for integrators not in possession of the NvM source code.⌋ ( )

10.2.2 NvM

SWS Item	**NVM539_Conf :**
Module Name	*NvM*
Module Description	Configuration of the NvM (NvRam Manager) module.

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
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.
NvmDemEventParameterRefs	0..1	Container for the references to DemEventParameter elements which shall be invoked using the API Dem_ReportErrorStatus API in case the corresponding error occurs. The EventId is taken from the referenced DemEventParameter's DemEventId value. The standardized errors are provided in the container and can be extended by vendor specific error references.

10.2.3 NvMCommon

SWS Item	**NVM028_Conf :**
Container Name	NvMCommon
Description	Container for common configuration options.
Configuration Parameters	

SWS Item	**NVM491_Conf :**	
Name	NvMApiConfigClass {NVM_API_CONFIG_CLASS}	
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.	
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM550_Conf :**		
Name	NvMBswMMultiBlockJobStatusInformation {NVM_BSWM_MULTI_BLOCK_JOB_STATUS_INFORMATION}		
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		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM492_Conf :**		
Name	NvMCompiledConfigId {NVM_COMPILED_CONFIG_ID}		
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	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM493_Conf :**		
Name	NvMCrcNumOfBytes {NVM_CRC_NUM_OF_BYTES}		
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	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	NVM494_Conf :		
Name	NvMDatasetSelectionBits {NVM_DATASET_SELECTION_BITS}		
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	--		
ConfigurationClass	Pre-compile time	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: ECU dependency: MemHwA, NVM_NVRAM_BLOCK_IDENTIFIER, NVM_BLOCK_MANAGEMENT_TYPE		

SWS Item	NVM495_Conf :		
Name	NvMDevErrorDetect {NVM_DEV_ERROR_DETECT}		
Description	Pre-processor switch to enable and disable development error detection. true: Development error detection enabled. false: Development error detection disabled.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	--		
ConfigurationClass	Pre-compile time	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	NVM496_Conf :		
Name	NvMDrvModeSwitch {NVM_DRV_MODE_SWITCH}		
Description	Preprocessor switch to enable switching memory drivers to fast mode during performing NvM_ReadAll and NvM_WriteAll true: Fast mode enabled. false: Fast mode disabled.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	--		
ConfigurationClass	Pre-compile time	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	NVM497_Conf :		
Name	NvMDynamicConfiguration {NVM_DYNAMIC_CONFIGURATION}		
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.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	--		
ConfigurationClass	Pre-compile time	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME

	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM498_Conf :**		
Name	NvMJobPrioritization {NVM_JOB_PRIORITIZATION}		
Description	Preprocessor switch to enable job prioritization handling true: Job prioritization handling enabled. false: Job prioritization handling disabled.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM500_Conf :**		
Name	NvMMultiBlockCallback {NVM_MULTI_BLOCK_CALLBACK}		
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	--		
maxLength	--		
minLength	--		
regularExpression	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM501_Conf :**		
Name	NvMPollingMode {NVM_POLLING_MODE}		
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	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM518_Conf :**	
Name	NvMRepeatMirrorOperations {NVM_REPEAT_MIRROR_OPERATIONS}	
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	

ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM502_Conf :**		
Name	NvMSetRamBlockStatusApi {NVM_SET_RAM_BLOCK_STATUS_API}		
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	--		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM503_Conf :**		
Name	NvMSizeImmediateJobQueue {NVM_SIZE_IMMEDIATE_JOB_QUEUE}		
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 .. 255		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module dependency: NVM_JOB_PRIORITIZATION		

SWS Item	**NVM504_Conf :**		
Name	NvMSizeStandardJobQueue {NVM_SIZE_STANDARD_JOB_QUEUE}		
Description	Defines the number of queue entries for the standard job queue.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	1 .. 255		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM505_Conf :**
Name	NvMVersionInfoApi {NVM_VERSION_INFO_API}
Description	Pre-processor switch to enable / disable the API to read out the modules version information [NVM285], [NVM286]. true: Version info API enabled. false: Version info API disabled.
Multiplicity	1
Type	EcucBooleanParamDef
Default value	--

ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency	scope: module		

No Included Containers

[NVM028] ⌈The following tables specify parameters that shall be definable in the module's configuration file (NvM_Cfg.h). ⌋ (BSW167, BSW00381, BSW00388, BSW00389, BSW00390, BSW00391, BSW00392, BSW00393, BSW00394, BSW00395, BSW00396, BSW00397, BSW171)

[NVM321] ⌈Pre-compile time configuration parameters to be implemented as "const" should be placed into a separate c-file (NvM_Cfg.c). ⌋ (BSW00419)

10.2.4NvMBlockDescriptor

SWS Item	**NVM061_Conf :**
Container Name	NvMBlockDescriptor
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	

SWS Item	**NVM476_Conf :**		
Name	NvMBlockCrcType {NVM_BLOCK_CRC_TYPE}		
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.	
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module dependency: NVM_BLOCK_USE_CRC, NVM_CALC_RAM_BLOCK_CRC		

SWS Item	**NVM477_Conf :**
Name	NvMBlockJobPriority {NVM_BLOCK_JOB_PRIORITY}
Description	Defines the job priority for a NVRAM block (0 = Immediate priority).
Multiplicity	1
Type	EcucIntegerParamDef

Range	0 .. 255		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM062_Conf :**		
Name	NvMBlockManagementType {NVM_BLOCK_MANAGEMENT_TYPE}		
Description	Defines the block management type for the NVRAM block.[NVM137]		
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.	
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM036_Conf :**		
Name	NvMBlockUseCrc {NVM_BLOCK_USE_CRC}		
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	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM519_Conf :**		
Name	NvMBlockUseSyncMechanism {NVM_BLOCK_USE_SYNC_MECHANISM}		
Description	Defines whether an explicit synchronization mechanism with a RAM mirror and callback routiness 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		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM033_Conf :**		
Name	NvMBlockWriteProt {NVM_BLOCK_WRITE_PROT}		
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	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM551_Conf :**		
Name	NvMBswMBlockStatusInformation {NVM_BSWM_BLOCK_STATUS_INFORMATION}		
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		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM119_Conf :**		
Name	NvMCalcRamBlockCrc {NVM_CALC_RAM_BLOCK_CRC}		
Description	Defines CRC (re)calculation for the permanent RAM block 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	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module dependency: NVM_BLOCK_USE_CRC		

SWS Item	**NVM116_Conf :**		
Name	NvMInitBlockCallback {NVM_INIT_BLOCK_CALLBACK}		
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	--		
maxLength	--		
minLength	--		
regularExpression	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	

Scope / Dependency	scope: module

SWS Item	**NVM533_Conf :**		
Name	NvMMaxNumOfReadRetries {NVM_MAX_NUM_OF_READ_RETRIES}		
Description	Defines the maximum number of read retries.		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 7		
Default value	0		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM499_Conf :**		
Name	NvMMaxNumOfWriteRetries {NVM_MAX_NUM_OF_WRITE_RETRIES}		
Description	Defines the maximum number of write retries for a NVRAM block with [NVM061_Conf]. 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	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM478_Conf :**		
Name	NvMNvBlockBaseNumber {NVM_NV_BLOCK_BASE_NUMBER}		
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	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module dependency: FEE_BLOCK_NUMBER, EA_BLOCK_NUMBER		

SWS Item	**NVM479_Conf :**
Name	NvMNvBlockLength {NVM_NV_BLOCK_LENGTH}
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 the sizeof

	operator). 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	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM480_Conf :**		
Name	NvMNvBlockNum {NVM_NV_BLOCK_NUM}		
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 NVM444. 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	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module dependency: NVM_BLOCK_MANAGEMENT_TYPE		

SWS Item	**NVM481_Conf :**		
Name	NvMNvramBlockIdentifier {NVM_NVRAM_BLOCK_IDENTIFIER}		
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		
Multiplicity	1		
Type	EcucIntegerParamDef (Symbolic Name generated for this parameter)		
Range	2 .. 65535		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module dependency: NVM_DATASET_SELECTION_BITS		

SWS Item	**NVM035_Conf :**
Name	NvMNvramDeviceId {NVM_NVRAM_DEVICE_ID}
Description	Defines the NVRAM device ID where the NVRAM block is located. Calculation Formula: value =

	TargetBlockReference.[Ea/Fee]BlockConfiguration.[Ea/Fee]DeviceIndex		
Multiplicity	1		
Type	EcucIntegerParamDef		
Range	0 .. 254		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module dependency: EA_DEVICE_INDEX, FEE_DEVICE_INDEX		

SWS Item	**NVM482_Conf :**		
Name	NvMRamBlockDataAddress {NVM_RAM_BLOCK_DATA_ADDRESS}		
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	--		
maxLength	--		
minLength	--		
regularExpression	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM521_Conf :**		
Name	NvMReadRamBlockFromNvCallback {NVM_READ_RAM_BLOCK_FROM_NVM}		
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	--		
maxLength	--		
minLength	--		
regularExpression	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM483_Conf :**
Name	NvMResistantToChangedSw {NVM_RESISTANT_TO_CHANGED_SW}
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	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM484_Conf :**		
Name	NvMRomBlockDataAddress {NVM_ROM_BLOCK_DATA_ADDRESS}		
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	--		
maxLength	--		
minLength	--		
regularExpression	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM485_Conf :**		
Name	NvMRomBlockNum {NVM_ROM_BLOCK_NUM}		
Description	Defines the number of multiple ROM blocks in a contiguous area according to the given block management type. 0-255 For NVRAM blocks to be configured of block management type NVM_BLOCK_DATASET. The actual range is limited according to NVM444. 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 .. 255		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module dependency: NVM_BLOCK_MANAGEMENT_TYPE, NVM_NV_BLOCK_NUM		

SWS Item	**NVM117_Conf :**		
Name	NvMSelectBlockForReadAll {NVM_SELECT_BLOCK_FOR_READALL}		
Description	NVM117: 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. 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	--		

ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module dependency: NVM_RAM_BLOCK_DATA_ADDRESS		

SWS Item	**NVM549_Conf :**		
Name	NvMSelectBlockForWriteAll {NVM_SELECT_BLOCK_FOR_WRITEALL}		
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. 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	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module dependency: NVM_RAM_BLOCK_DATA_ADDRESS		

SWS Item	**NVM506_Conf :**		
Name	NvMSingleBlockCallback {NVM_SINGLE_BLOCK_CALLBACK}		
Description	Entry address of the block specific callback routine which shall be invoked on termination of each asynchronous single block request [NVM113].		
Multiplicity	0..1		
Type	EcucFunctionNameDef		
Default value	--		
maxLength	--		
minLength	--		
regularExpression	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM532_Conf :**		
Name	NvMStaticBlockIDCheck {NVM_STATIC_BLOCK_ID_CHECK}		
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		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM072_Conf :**
Name	NvMWriteBlockOnce {NVM_WRITE_BLOCK_ONCE}
Description	Defines write protection after first write. The NVRAM manager sets the write protection bit after the NV block was written the first time. This means that some of the NV blocks in the

	NVRAM should never be erased nor be replaced with the default ROM data after first initialization. [NVM276]. true: Defines write protection after first write is enabled. false: Defines write protection after first write is disabled.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM520_Conf :**		
Name	NvMWriteRamBlockToNvCallback {NVM_WRITE_RAM_BLOCK_TO_NVM}		
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	--		
maxLength	--		
minLength	--		
regularExpression	--		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM534_Conf :**		
Name	NvMWriteVerification {NVM_WRITE_VERIFICATION}		
Description	Defines if Write Verification is enabled. false: Write verification is disabled. true: Write Verification is enabled.		
Multiplicity	1		
Type	EcucBooleanParamDef		
Default value	false		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

SWS Item	**NVM538_Conf :**		
Name	NvMWriteVerificationDataSize {NVM_WRITE_VERIFICATION_DATA_SIZE}		
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 .. 65536		
Default value	0		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency	scope: module		

Included Containers		
Container Name	***Multiplicity***	***Scope / Dependency***
NvMTargetBlockReference	1	This parameter is just a container for the parameters for EA and FEE

10.2.5NvMTargetBlockReference

SWS Item	**NVM486_Conf :**
Choice container Name	NvMTargetBlockReference
Description	This parameter is just a container for the parameters for EA and FEE

Container Choices		
Container Name	***Multiplicity***	***Scope / Dependency***
NvMEaRef	0..1	EEPROM Abstraction
NvMFeeRef	0..1	Flash EEPROM Emulation

10.2.6NvMEaRef

SWS Item	**NVM487_Conf :**
Container Name	NvMEaRef
Description	EEPROM Abstraction
Configuration Parameters	

SWS Item	**NVM488_Conf :**		
Name	NvMNameOfEaBlock		
Description	reference to EaBlock		
Multiplicity	1		
Type	Reference to [EaBlockConfiguration]		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency			

No Included Containers

10.2.7NvMFeeRef

SWS Item	**NVM489_Conf :**
Container Name	NvMFeeRef
Description	Flash EEPROM Emulation
Configuration Parameters	

SWS Item	**NVM490_Conf :**
Name	NvMNameOfFeeBlock
Description	reference to FeeBlock

Multiplicity	1		
Type	Reference to [FeeBlockConfiguration]		
ConfigurationClass	***Pre-compile time***	X	VARIANT-PRE-COMPILE
	Link time	X	VARIANT-LINK-TIME
	Post-build time	--	
Scope / Dependency			

No Included Containers

10.2.8NvmDemEventParameterRefs

SWS Item	**NVM541_Conf :**
Container Name	NvmDemEventParameterRefs
Description	Container for the references to DemEventParameter elements which shall be invoked using the API Dem_ReportErrorStatus API in case the corresponding error occurs. The EventId is taken from the referenced DemEventParameter's DemEventId value. The standardized errors are provided in the container and can be extended by vendor specific error references.
Configuration Parameters	

SWS Item	**NVM542_Conf :**		
Name	NVM_E_INTEGRITY_FAILED		
Description	Reference to the DemEventParameter which shall be issued when the error "API request integrity failed" has occured.		
Multiplicity	0..1		
Type	Reference to [DemEventParameter]		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

SWS Item	**NVM546_Conf :**		
Name	NVM_E_LOSS_OF_REDUNDANCY		
Description	Reference to the DemEventParameter which shall be issued when the error "loss of redundancy" has occured.		
Multiplicity	0..1		
Type	Reference to [DemEventParameter]		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

SWS Item	**NVM547_Conf :**		
Name	NVM_E_QUEUE_OVERFLOW		
Description	Reference to the DemEventParameter which shall be issued when the error "NVRAM Managers job queue overflow" has occured.		
Multiplicity	0..1		
Type	Reference to [DemEventParameter]		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	

Scope / Dependency	

SWS Item	**NVM543_Conf :**		
Name	NVM_E_REQ_FAILED		
Description	Reference to the DemEventParameter which shall be issued when the error "API request failed" has occured.		
Multiplicity	0..1		
Type	Reference to [DemEventParameter]		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

SWS Item	**NVM545_Conf :**		
Name	NVM_E_VERIFY_FAILED		
Description	Reference to the DemEventParameter which shall be issued when the error "Write Verification failed" has occured.		
Multiplicity	0..1		
Type	Reference to [DemEventParameter]		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

SWS Item	**NVM548_Conf :**		
Name	NVM_E_WRITE_PROTECTED		
Description	Reference to the DemEventParameter which shall be issued when the error "write attempt to NVRAM block with write protection" has occured.		
Multiplicity	0..1		
Type	Reference to [DemEventParameter]		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

SWS Item	**NVM544_Conf :**		
Name	NVM_E_WRONG_BLOCK_ID		
Description	Reference to the DemEventParameter which shall be issued when the error "Static Block ID check failed" has occured.		
Multiplicity	0..1		
Type	Reference to [DemEventParameter]		
ConfigurationClass	***Pre-compile time***	X	All Variants
	Link time	--	
	Post-build time	--	
Scope / Dependency			

No Included Containers

10.3 Common configuration options

[NVM030] ⌈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.⌋ ( )

[NVM034] ⌈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 must 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. ⌋(BSW135)

[NVM073] ⌈The comparison between the stored configuration ID and the compiled configuration ID shall be done as the first step within the function NvM_ReadAll during startup. ⌋ ( )

[NVM688] ⌈In case of a detected configuration ID mismatch, the behavior of the NvM module shall be defined by a configurable option. ⌋ ( )

[NVM052] ⌈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). ⌋ ( )

10.4 Published parameters

[NVM743] ⌈The standardized common published parameters as required by BSW00402 in the General Requirements on Basic Software Modules 3 shall be published within the header file of this module and need to be provided in the BSW Module Description. The according module abbreviation can be found in the List of Basic Software Modules 3. ⌋ (BSW00402, BSW00374, BSW00379, BSW003, BSW00318)

Additional module-specific published parameters are listed below if applicable.

11 Changes to Release 4.0 Rev 2

11.1 Deleted SWS Items

SWS Item	*Rationale*
NVM540_Conf	Removed NvMMainFunctionCycleTime

11.2 Replaced SWS Items

SWS Item	*replaced by SWS Item*	*Rationale*

11.3 Changed SWS Items

SWS Item	*Rationale*
NVM538_Conf	Updated NvMWriteVerificationDataSize range
NVM478_Conf	Updated NvMNvBlockBaseNumber description
NVM158	NvM_ReadAll called by ECUM or integration code
NVM225	NvM_CancelJobs behaviour
NVM734	Corrected inconsistency between C-interface and port interface
NVM737	
NVM738	
NVM258	Interruption of an ongoing job by an immediate priority request
NVM560	NvM_SetBlockLockStatus API configuration class
NVM561	
NVM562	
NVM455	Fixed input parameters and return values that complex types shall be passed/returned as pointer to constant
NVM539	
NVM540	

11.4 Added SWS Items

SWS Item	*Rationale*
NVM550_Conf	NvM and BswM interaction
NVM551_Conf	
NVM745	
NVM746	
NVM747	NvM_SetBlockLockStatus API functional description
NVM748	
NVM749	
NVM750	
NVM751	
NVM752	
NVM753	
NVM754	
NVM755	Include structure

12 AUTOSAR Service implemented by the NVRAM Manger

12.1 Scope of this Chapter

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.

12.2 Overview

12.2.1 Architecture

In the AUTOSAR ECU architecture (see [2]) the NvM module implements an AUTOSAR service.
From the viewpoint of the basic software module “NVRAM Manager”, there are three kinds of dependencies between the service and the AUTOSAR software components above the RTE[1] :

- the application accesses the API (implemented as C-functions) of the NvM module
- the application is optionally notified upon the outcome of requested asynchronous activity (via callback-C-functions by the NvM module)
- the application accesses the RAM-blocks which the NvM module has to save or restore

These dependencies must be described in terms of the AUTOSAR meta-model which will contribute to the SW-C Description of the application component as well as to the SW-C Description of the NVRAM Service.

12.2.2 Requirements

There are three sources of requirements for this specification:

- The requirements for the functionality of the NVRAM service are specified in [4].
- In order to model the VFB view of the Service, the chapter on AUTOSAR Services of the VFB specification [9] has to be considered as an additional requirement.
- For the formal description of the SW-C attributes [10] gives the requirements.

[1] “Applications” of the NVRAM Manager can be “below” as well as “above” the RTE; this chapter concentrates on the interfaces seen from the applications above the RTE.

12.2.3Use Cases

On each ECU we have typically one instance of the NVRAM Service and several Atomic Software Component instances, named “clients” further on in this chapter, which are using this Service. In addition, there are parts of the basic software, which either control the NvM module (e.g. for initiation and shutdown) or need to read or write some NVRAM data themselves.

Each client will own certain transient data which it “wants” to make persistent via the NVRAM Service. It is important to mention that the transient data are privately owned by clients[2] – otherwise, the clients could communicate via the content of those data which is against AUTOSAR principles. Towards the NVRAM Service, the client uses so-called block identifiers to address the persistent data.

Furthermore, it is important that in general a client is a component instance. So if a client component can be instantiated more than once on an ECU, each instance will in general possess its own private copy of transient data. This is an important restriction on the modeling of block identifiers: Because for a reusable component, we cannot a priori prescribe the number of instances. We can use block identifiers (which could be in the form of symbolic information) in the code only on a per-component base, but this must result in different actual identifiers towards the NVRAM.

There are three principle ways, how a client can use the NVRAM service.

12.2.3.1 Implicit Update and Restore of RAM Mirror

See Figure 21. This is a summary of use cases in which there is no communication (in the sense of the VFB) between the NvM module and its client. The client holds its transient data in a so-called RAM mirror which is organized in so-called RAM blocks permanently associated with their non-volatile counterparts of the NvM module. Update and restore of the RAM blocks is initiated by the ECU State Manager, the AUTOSAR SW-C accesses its RAM blocks directly. Note that the NvM module does NOT communicate with its client via the RAM blocks because no information is exchanged between the NvM module and its client via the RAM block.
In this case, the client does not need any AUTOSAR Interface to communicate with the Service. But nevertheless, there have to be certain contracts between the client and the NvM module:
On the size, identification and maybe further properties of the RAM/NVRAM blocks which have to be known on both sides. For the access of the RAM mirror via the RTE see 12.4.
On the states (of the ECU and/or the client component) in which the RAM blocks can be safely accessed by the client. For this contract, it is essential that the client is informed about its activation/termination or about the relevant ECU states. This communication path has to be handled by the RTE in cooperation with the ECU State Manager (not visible in Figure 21).

[2] If the client is an Atomic SWC. It does not hold for components of the basic SWC which might use the NVRAM Manager.

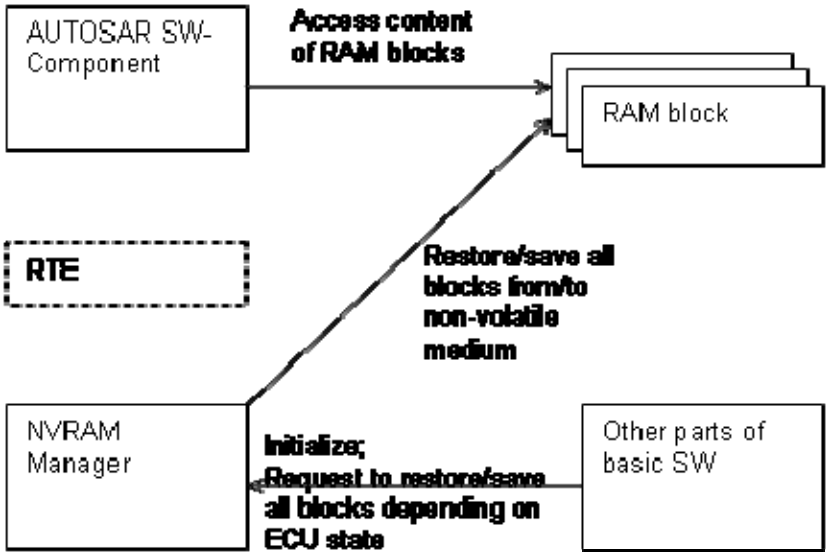

Figure 21: Implicit update/restore of RAM blocks

12.2.3.2 Explicit Update and Restore of RAM Mirror

See Figure 22. This is a summary of use cases in which the client explicitly requests to save or restore the content of a mirror RAM block from/to non-volatile memory (e.g. in order to decrease the risk of data loss). But as in the first case, RAM blocks are permanently associated with their counterparts in NVRAM. Now we need direct communication (in the sense of the VFB) between the NvM module and its client. Also in this case, the NvM module does NOT communicate with its client via the RAM blocks. In this case, the client must use an AUTOSAR interface to communicate with the service.

Similar to the case of implicit update/restore, we need certain contracts between the client and the NvM module:

On the size, identification and maybe further properties of the RAM/NVRAM blocks which have to be known on both sides. For the access of the RAM mirror via the RTE see 12.4.

On the states (of the ECU and/or the client component) in which the client may request to restore or save RAM blocks. For this contract, it is essential that the client is informed about its activation/termination or relevant ECU states. This communication path has to be handled by the RTE in cooperation with the ECU State Manager (not visible in Figure 22).
During the explicit update and restore scenarios, the client has to follow certain rules, e.g. not to access a RAM block before the update or restore is finished.

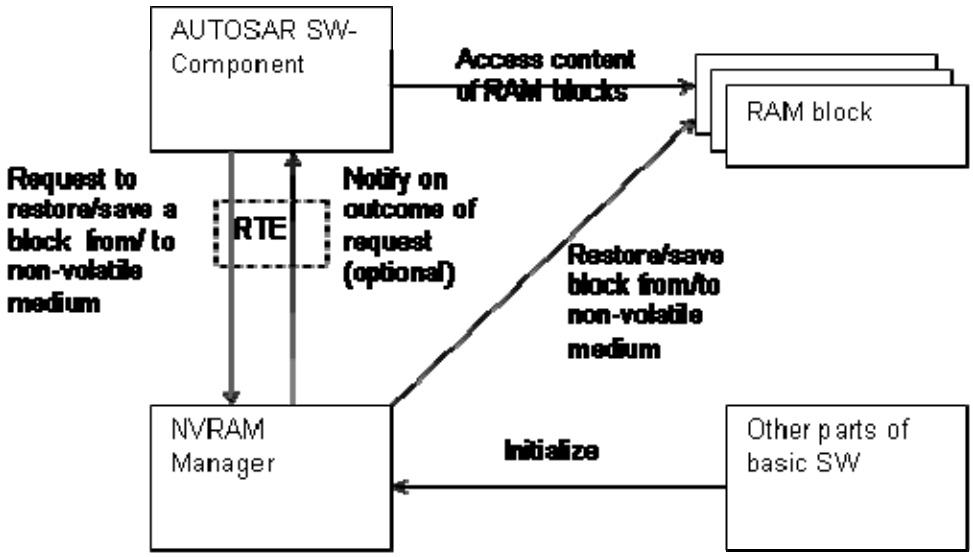

Figure 22: Explicit update/restore of mirror RAM blocks

12.2.3.3 Explicit Update and Restore via a Local Buffer

See Figure 23. This is a summary of use cases in which the client explicitly requests to save or restore the content of some transient data, but instead of using a RAM mirror, it passes the address of a buffer which is only temporarily associated with a certain block in NVRAM. Also, we need here direct communication (in the sense of the VFB) between the NvM module and its client. Also in this case, the NvM module does NOT communicate with its client via the RAM blocks.
Also in this case, the client must use an AUTOSAR interface to communicate with the service.

Again we need certain contracts between the client and the NvM module:

- On the size, identification and maybe further properties of the temporarily associated NVRAM blocks which have to be known on both sides. For the buffer, the partners do not have to agree on its allocation because it is totally up to the client.
- On the states (of the ECU and/or the client component) in which the client may request to restore or save RAM blocks. For this contract, it is essential that the client is informed about its activation/termination or relevant ECU states. This communication path has to be handled by the RTE in cooperation with the ECU State Manager (not visible in Figure 23).
- During the explicit update and restore scenarios, the client has to follow certain rules, e.g. not to access the buffer before the update or restore is finished.

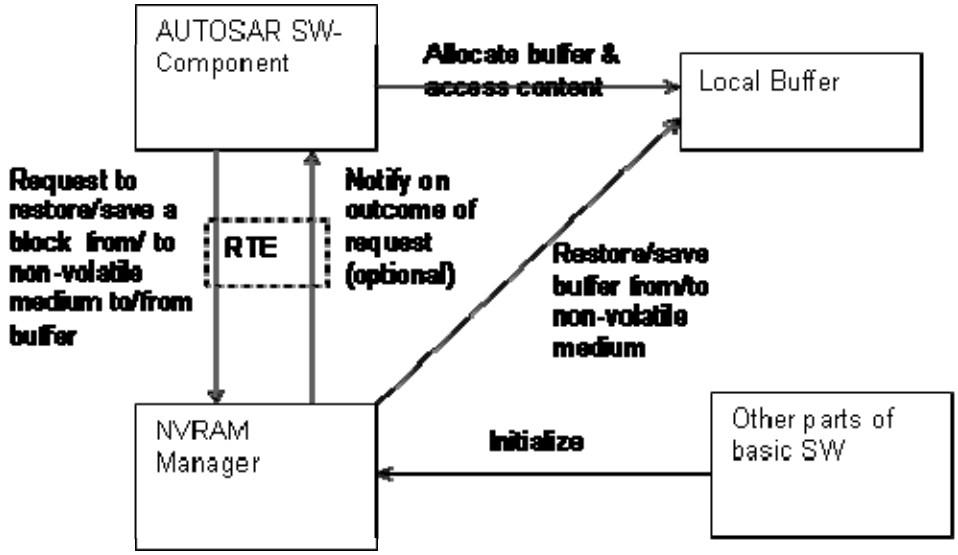

Figure 23: Explicit update/restore of RAM blocks via buffer

12.3 Specification of the Ports and Port Interfaces

This chapter specifies the ports and port interfaces which are needed in order to operate the NvM module functionality over the VFB. Note that there are ports on both sides of the RTE: The SW-C description of the NVRAM Service defines the ports below the RTE. Each SW-Component which uses the Service must contain “service ports” in its own SW-C description which are connected to the ports of the NvM module, so that the RTE can be generated.

12.3.1 Ports and Port Interface for Single Block Requests

12.3.1.1 General Approach

It is appropriate to model the requests issued from a client to the NVRAM Service by ports with client/server interfaces.
Typically, a client of the application domain needs the NvM module for services dealing with individual blocks (except for the simple use case described in 12.2.3.1). These so-called single block requests of the NvM module C-API need the Block ID as a first argument for the C-function.
In order to keep the client code independent from the configuration of NVRAM block IDs (which depend on the needs of all the applications on an ECU), the block IDs are not passed from the clients to the NvM module, but are modeled as “port defined argument values” of the Provide Ports on the NvM module side. As a consequence, the block IDs will not show up as arguments in the operations of the client-server interface. As a further consequence of this approach, there will be separate ports for each NVRAM block both on the client side as well as on the server side.

12.3.1.2 Data Types

This chapter describes the data types which will be used in the port interfaces for single block requests and notifications.
The data types uint8 and boolean used in the interfaces refer to the basic AUTOSAR data types.
The data type NvM_RequestResultType indicates the result of a read or write request which are defined by the SW-C Description as follows:

```
ImplementationDataType NvM_RequestResultType {
     category TYPE_REFERENCE
     ImplementationDataType uint8
};
```

Via an associated CompuMethod the following constants are defined within this type:

0 -> NVM_REQ_OK
1 -> NVM_REQ_NOT_OK
2 -> NVM_REQ_PENDING
3 -> NVM_REQ_INTEGRITY_FAILED
4 -> NVM_REQ_BLOCK_SKIPPED
5 -> NVM_REQ_NV_INVALIDATED
6 -> NVM_REQ_CANCELED
7 -> NVM_REQ_REDUNDANCY_FAILED
8 -> NVM_REQ_RESTORED_FROM_ROM

12.3.1.3 Port Interface

All single block operations (with the exception of SetBlockProtection which is discussed in chapter 12.3.3) are put into one single port interface in order to minimize the number of ports and names needed in the XML description.

The operations correspond to the function calls of the NVRAM C-API (notation in pseudo code; must be transferred into XML). Compared to the C-function, we do not need the “NvM_” prefix in the names because the names given here will show up in the XML not as global entities but as part of an interface description.
The notation of possible error codes resulting from server calls follows the approach in the meta-model. It is a matter of the RTE specification [11] how those error codes will be passed via the actual API.

[NVM734] ⌈

```
ClientServerInterface NvMService{
     PossibleErrors {
      E_NOT_OK = 1
     };
     // The next operation is always provided
     GetErrorStatus(    OUT    NvM_RequestResultType    RequestResultPtr    ,
ERR{E_NOT_OK} );

     // The next two operations are always available, but are needed
     // only for “data set” block types. Thus they shall be provided in
     // the interface only for those block types
     SetDataIndex( IN uint8 DataIndex , ERR{E_NOT_OK} );
     GetDataIndex( OUT uint8 DataIndexPtr , ERR{E_NOT_OK} );
```

```
    // This operation is only provided via optional configuration
    // NvMSetRamBlockStatusApi
    SetRamBlockStatus( IN boolean BlockChanged , ERR{E_NOT_OK} );

    // The next three operations are only provided for
    // NVRAM API configuration class 2 and 3
    ReadBlock ( IN void DATA_REFERENCE DstPtr, ERR{E_NOT_OK} );
    WriteBlock ( IN void DATA_REFERENCE SrcPtr, ERR{E_NOT_OK} );
    RestoreBlockDefaults ( IN void DATA_REFERENCE DstPtr, ERR{E_NOT_OK} );

    // The next two operations are only provided for
    // NVRAM API configuration class 3
    EraseBlock( ERR{E_NOT_OK} );
    InvalidateNvBlock( ERR{E_NOT_OK} );
};
⌋()
```

Note: WriteBlock may modify the CRC, hence the INOUT. Note that in the interface of each Require Port of the client, only those operations must be present which the client actually requires for that block.
Note: DstPtr and SrcPtr are pointers of ImplementationDataType void.

On the other hand, in the interface of the Provide Port, only those operations will be provided, which actually have to be configured for the NVRAM Service on a specific ECU. Because some of the operations are optional, they may not always be provided. The optional parts are indicated by comments in the pseudo-code and must currently be configured manually in the XML Specification3.

Note that via compatibility rules, a Require Port can be connected to a Provide Port providing more operations than actually needed, but not the other way round.

Note also, that due to the optional parts, different variants of this interface may exist on the same ECU. In this case, it may be required to make the interface name unique by attaching suffixes to its name. A rule for this is however not standardized.

12.3.1.4 Ports

Figure 24 shows how AUTOSAR Software components (single or multiple instances) are connected by means of service ports to the NvM module.

3 By introducing a property concept in the meta-model, the optional parts may in future be controlled by other model elements. For this, the relation between the Service requirements of an SWC and the Service ports of an SWC must be explicitly modeled.

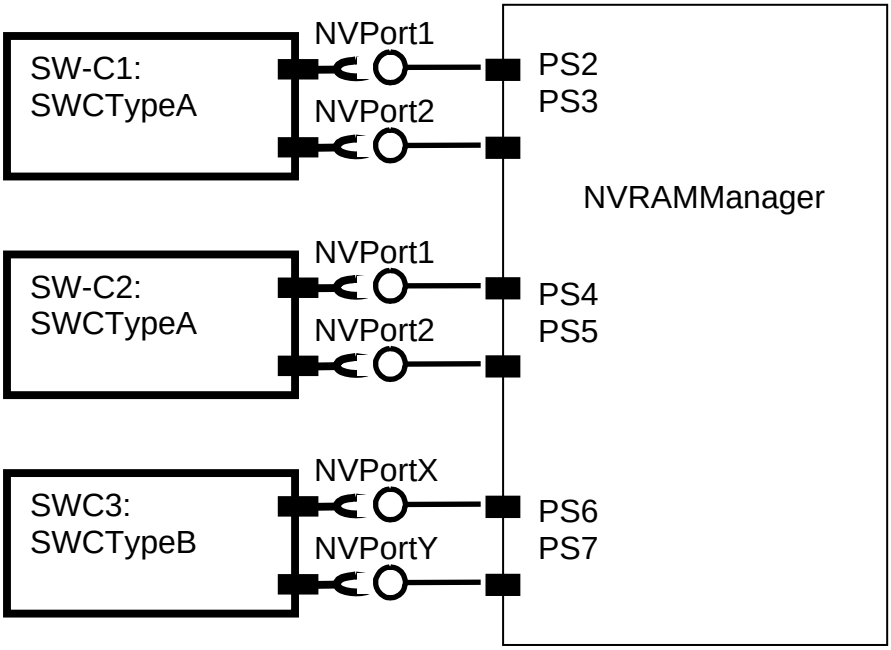

Figure 24: Example of SW-Cs connected to the NVRAM via service ports.

On the left side, there are two instances of component SWCTypeA and one instance of component SWCTypeB. The Port names on the left side are only for illustration. No notification ports or administrative ports are configured.
On the NvM module side, there is one port per NVRAM block providing all the services of the interface NvMService described above. Each client has one port for requiring those services for each NVRAM block associated with that client.
The ports providing the services are named according to the BlockID:
PS2, PS3, … ,PS7
These names are examples, they are not standardized. It is not essential, that these ports are numbered consecutively, but the numbers should match the NVRAM blocks for documentation purposes.
Hint for the developers of the client components:
There are two ways of accessing the NvM Service ports via the RTE. The selection of one alternative is an implementation decision independent of the specification of the NVRAM Service.

- Use the “direct API” of the RTE. In the client code, each single block operation will show up as a separate call to the RTE which can be optimized to “zero overhead” function invocation under certain conditions (source code of the client is available and the client is a single-instance component). In case of optimization, there will be a macro for each operation and each block. Each macro will be expanded to a direct call of the NvM module.
- Use the “indirect API” of the RTE. The client code will call the single block operations via a “port handle”. This port handle can be accessed by an array index which in this case identifies the NVRAM blocks used by this client. This approach offers less potential for code size optimization in the RTE, but the port handle allows for a more compact handling in the client code, e.g. if iteration over NVRAM blocks is required. Note that the ports accessed via this

kind of API must have the same port interface, i.e. the optional parts must be the same.

For more explanation of the direct and indirect API see [11].

12.3.2Ports and Port Interface for Notifications

Some block requests which can be initiated by client requests are performed asynchronously (with respect to the requesting call) and can be configured to throw “notifications” to indicate the completion (or error) of such a request. The reason is the relatively slow access time of the storage medium.

As a first glance a pair of request/notification looks like an asynchronous client/server communication described in [9], chapter 4.4. But it cannot be modeled that way, because in an asynchronous client/server communication, the server operation in itself is always synchronous whereas for the NvM module, the request of an action, the actual performing of it and the notification are asynchronous activities. Another reason is that the notifications are optional.

Therefore it is required to model the notification by a separate connection and thus separate ports on both service and SW component side.

The NvM module on C-code level does not pass a block ID with the callback but allows configuring a separate callback address for each block. This means that on the modeling level we need one separate (optional) notification port for each single block. As a consequence, the existence of notification ports in the SW-C Description of the NVRAM Service must be configured per ECU.

For the notification ports, a client/server interface is used, because we have to transmit two values. A drawback of this approach (in comparison to sender/receiver) is that a badly implemented SW-C could block the NvM module for an unacceptable amount of time. To minimize this risk there are two possibilities:
Design rules for the implementation of such notifications by the SW-Cs
Configuring the notification interface as asynchronous client/server would allow decoupling of the invocation in the SW-C from the implementation of the notification function. As this involves a more complicated protocol, this shall however be used only in special cases and is not part of the standardization of this Service.
The notification has to pass the following arguments:
A “Service ID” which indicates which one of the asynchronous services triggered via the operations of Interface NvMService (see above) the notification belongs to.
A JobRequestResult indicating success or failure of the service.
The Pseudo-Code for the notification interface (which has to be translated into XML) is:

[NVM735] ⌈

```
ClientServerInterface NvMNotifyJobFinished {
     JobFinished( IN uint8 ServiceId, IN NvM_RequestResultType JobResult);
};
```

⌋ ( )

The Pseudo-Code for the init block notification interface (which has to be translated into XML) is:

[NVM736] ⌈
```
ClientServerInterface NvMNotifyInitBlock {
     InitBlock();
};
```
⌋()
It is recommended to name the ports providing the notifications via this interface according to the BlockID (but this is no standard), for example:
PN2, PN3,…
Interrupt context: „This routine might be called in interrupt context, depending on the calling function.“ (see Chapter 0).
This requirement is consistent with current RTE requirements if the interrupt context is not propagated ”above“ the RTE. This must be handled by the RTE generation. Currently (in AUTOSAR 2.1), it is not possible to indicate this condition in the SWC Description of the Service. As a workaround, the runnable implementing the callback in the client component must be marked in this case as “canBeInvokedConcurrently = FALSE”. Note that this is only a workaround because this condition should be handled independently of the SWC description of the client components.

12.3.3Ports and Port Interfaces for Administrative Operations

Administrative functions which are not needed for “normal” use cases are put into a separate port interface. Currently it contains only the operation “SetBlockProtection”. The port interface is defined as follows. It does not specify error codes:

[NVM737] ⌈
```
ClientServerInterface NvMAdmin {
 PossibleErrors {
  E_NOT_OK = 1
 };
     // the next operation is only provided for NVRAM API configuration
     // class 3; besides of that it should only be required by a client
     // for special use cases, e.g. if a block protection has to be
     // removed during initiating of EOL data
     SetBlockProtection( IN boolean ProtectionEnabled, ERR{E_NOT_OK} );
};
```
⌋()
For the purpose of this document, ports providing this interface have the name PAdmin<nn> for a port identifier <nn>.

12.3.4Ports and Port Interfaces for Mirror Operations

The mirror functions provided by the client (to be called by the NvM module) and described in Chapter 7.2.2.17 are summarized here:

[NVM738] ⌈
```
ClientServerInterface NvMMirror {
 PossibleErrors {
  E_NOT_OK = 1
 };
   ReadRamBlockFromNvm ( IN void DATA_REFERENCE SrcPtr, ERR{E_NOT_OK} );
   WriteRamBlockToNvm ( IN void DATA_REFERENCE DstPtr, ERR{E_NOT_OK} );
}
```
⌋()
Note: The asynchronous operations ReadBlock and WriteBlock may not work on partitioned systems.

12.3.5Summary of all Ports

We end up with the following structure for the AUTOSAR Interface of the NvM module:
For access from “normal” application components above the RTE:

- For each memory block:
  - One port with a client-server interface providing all block-related services for that block (except SetBlockProtection).
  - Optional: A port with a client-server interface requiring a callback related to notifications for that block.

For administrative purposes from above the RTE:

- For each memory block:
  - Optional: One port with a client-server interface providing administrative services for that block

We indicate this as follows in pseudo code:

```
Service NvM
{
     // the entries in the next section will be defined only, if they
     // are actually required by client service ports
     ProvidePort NvMService<Block ID 1> PS2;
     ProvidePort NvMService<Block ID 2> PS3;
     ProvidePort NvMService<Block ID 3> PS4;
     …
     // the entries in the next section will be defined only, if the
     // notification sinks are actually provided by client service ports
     RequirePort NvMNotifyJobFinished PN2;
     RequirePort NvMNotifyJobFinished PN3;
     RequirePort NvMNotifyInitBlock PN4;
     …
     // the entries in the next section will be defined only, if they
     // are actually required by client service ports, i.e. in special
     // cases, where SetBlockProtection is needed
     ProvidePort NvMAdmin PAdmin2;
     ProvidePort NvMAdmin PAdmin3;
```

```
    ProvidePort NvMAdmin PAdmin4;
    …
    // the entries in the next section will be defined only, in case of
    // explicit synchronization
    RequirePort NvMMirror PM2;
    RequirePort NvMMirror PM3;
    RequirePort NvMMirror PM4;
    …
};
```

It is obvious that the existence of all these port definitions depends on the ECU. But also the port interfaces itself are in general ECU dependent because they contain optional parts as shown before. But they consist of standardized elements (operation prototypes, data types).

12.4 Access to the Memory Blocks

The NvM module does not specify how its clients actually access the RAM block. This is the private responsibility of each client. This especially holds true for the structure and semantics of the block content.
However, the current meta model allows the SW-C to specify the overall features of its NVRAM blocks (identifier, size, criticality, existence of ROM defaults etc.). For details see the specification of NvBlockNeeds in the SoftwareComponent Template. The NVRAM service has to be configured in response to those configurations given in the SW-C descriptions of its clients. The configuration parameters of the NVRAM Service are however more concrete (it has for example a “block management type”) whereas the SW-C template defines the requirements on such a service in a more feature-related way.
The following is only relevant if the client uses a “RAM mirror” (use cases 12.2.3.1 and 12.2.3.2): From the viewpoint of the client, each block of its RAM mirror can be modeled as a PerInstanceMemory. The RTE will then handle static allocation of those memory regions. In order to get access to one of its memory blocks in RAM, the client has to use the RTE API: “Rte_Pim” (see [11]).
By this, the client gets a correctly typed handle to the memory section. There is no need to standardize the name of the memory section because the RTE expands it with the component and/or instance name in order to avoid name clashes.
Note the start address and size of the C data object associated with the PerInstanceMemory sections must be harmonized with the configuration of the NVRAM block descriptors in the ECU configuration description. This issue is not part of this Specification.

12.5 Internal Behavior

The NvM module specification does not standardize the basic type to be used for identifying the NVRAM blocks since the needed binary size is ECU dependent. This type has to be defined for a specific ECU as follows:

```
ImplementationDataType NvM_BlockIdType {
    category TYPE_REFERENCE
    ImplementationDataType uint16
};
```

This type does not show up in the service ports of the client components because the block identifier is implemented as port defined argument value (see chapter 12.5) which is part of the InternalBehavior of the NVRAM Service. So the ECU dependency of NvM_BlockIdType is not visible for the clients.
Values 0 and 1 have special internal meaning and must not be used as identifiers for “normal” NVRAM blocks.
The InternalBehavior of the NVRAM Service is only seen by the local RTE. Besides the definition of the block identifiers as port defined arguments, it must specify the operation invoked runnables:

```
SwcInternalBehavior NVRAMManager {
// definition of associated operation-invoked RTE-events not shown
// (it is done in the same way as for any SWC type)
// section “runnable entities”:
RunnableEntity GetErrorStatus
     symbol “NvM_GetErrorStatus”
     canbeInvokedConcurrently = TRUE

RunnableEntity SetDataIndex
     symbol “NvM_SetDataIndex”
     canbeInvokedConcurrently = TRUE

RunnableEntity GetDataIndex
     symbol “NvM_GetDataIndex”
     canbeInvokedConcurrently = TRUE

RunnableEntity SetRamBlockStatus
     symbol “NvM_SetRamBlockStatus”
     canbeInvokedConcurrently = TRUE
RunnableEntity ReadBlock
     symbol “NvM_ReadBlock”
     canbeInvokedConcurrently = TRUE
RunnableEntity WriteBlock
     symbol “NvM_WriteBlock”
     canbeInvokedConcurrently = TRUE
RunnableEntity RestoreBlockDefaults
     symbol “NvM_RestoreBlockDefaults”
     canbeInvokedConcurrently = TRUE
RunnableEntity EraseNvBlock
     symbol “NvM_EraseNvBlock”
     canbeInvokedConcurrently = TRUE
RunnableEntity InvalidateNvBlock
     symbol “NvM_InvalidateNvBlock”
     canbeInvokedConcurrently = TRUE
RunnableEntity SetBlockProtection
     symbol “NvM_SetBlockProtection”
     canbeInvokedConcurrently = TRUE

// for each port providing the NvMService Interface:
PortDefinedArgumentValue {port=PS2, valueType=NvM_BlockIdType, value=2}

PortDefinedArgumentValue {
     port=PS<nn>,
     valueType=NvM_BlockIdType,
     value=<nn>}

// for each port providing the NvMAdministration Interface:
PortDefinedArgumentValue {
```

```
    port=PAdmin<xx>,
    valueType=NvM_BlockIdType,
    value=<xx>}
    …
    // end of section “runnable entities”
};
```

12.6 Configuration of the Block IDs

The Block IDs of the NvM module are modeled as “port defined argument values”. Thus the configuration of those values is part of the input to the RTE generator. Pre-compile configuration can be done by changing the XML specification for the argument values on the NVRAM Service at ECU integration time. Note that the ports visible on the client side are not affected by this.

13 Not applicable requirements

[NVM744] ⌈These requirements are not applicable to this specification.⌋ (BSW00344, BSW00404, BSW00405, BSW170, BSW00380, BSW00412, BSW00398, BSW00399, BSW00400, BSW00416, BSW168, BSW00423, BSW00426, BSW00427, BSW00431, BSW00432, BSW00434, BSW00375, BSW00422, BSW00420, BSW00417, BSW00336, BSW161, BSW162, BSW00324, BSW005, BSW00415, BSW164, BSW00325, BSW00326, BSW00342, BSW00343, BSW160, BSW007, BSW00347, BSW00307, BSW00335, BSW00314, BSW00348, BSW00353, BSW00361, BSW00302, BSW00328, BSW00312, BSW006, BSW00304, BSW00355, BSW00378, BSW00306, BSW00308, BSW00309, BSW00371, BSW00330, BSW009, BSW010, BSW00321, BSW00341, BSW00334, BSW130)