VECTOR

MICROSAR NVM

Technical Reference

Version 11.03.00

Authors	virck, viroto, vircre, virbka, virsrl, virstl, virlra, virljs, vireno, virbmz
Status	Released

Document Information

History

Author	Date	Version	Remarks
virck	2007-08-20	1.4	AUTOSAR 2.1, updated for EAD3.1 usage, conversion to new template
virck	2007-12-06	3.01.00	Change of the document's versioning scheme to correspond to the module's major and minor, update of parameter description in chapter 'Graphical Configuration of NvM' and service port generation description, remove of DATASET ROM, feature not supported anymore, remove of introduction paragraphs from 'Description of Memory Mapping and Compiler Abstraction', not subject of this document, simplified 'Block Management Types' naming, formal changes
virck	2008-01-11	3.01.01	New chapter to clarify the dependency on the CRC library, stated explicitly that DET is optional, corrected default values
Manfred Duschinger, Heike Bischof	2008-05-23	3.02.00	AUTOSAR 3, conversion to Technical Reference
Manfred Duschinger	2008-12-08	3.03.00	ESCAN00027300: Description of `NvM_ServiceIdType` in SingleBlockCallbackFunction and MultiBlockCallbackFunction Description and expected caller context of `NvM_SetBlockLockStatus`-API reworked. Chapter 4.4.17 ‘Concurrent access to NV data for DCM’ added. Chapter 4.4.5.2: Write order at redundant blocks added. Expansion of glossary. Chapter 7.2.2: Description of ‘Dataset Selection Bits’ added.
Manfred Duschinger	2009-02-25	3.03.01	ESCAN00031177: Manufacturer specific requirements attribute for traceability reasons
Manfred Duschinger	2009-03-24	3.03.02	ESCAN00032480: Missing information in documentation:

			Chapter 6.4.5: ‘Description of NvM_RequestResultType added’. Chapters 6.4.15 and 6.4.16: ‘Services are multiblock requests’.
Manfred Duschinger	2009-06-03	3.04.00	ESCAN00032480: Update of History of version 3.03.02: Updated changed chapters. Chapter 6.2: ‘Block ID 0 is only allowed for API NvM_GetErrorStatus()’ ESCAN00033075: Chapter 4.5.1.1: DataIndex Check in NvM_ReadBlock() added. DataIndex Check was also added to NvM_InvalidateNvBlock() and NvM_EraseNvBlock(). ESCAN00033900: Chapter 4.4.17: “Priority Handling of DCM-Blocks” ESCAN00035089: Chapters 4.1, 7.2.2 “Callbacks NvM_JobEndNotification, NvM_JobErrorNotification implemented” ESCAN00034073: Chapters 2, 4.4.5.1, 7.2.2 “Crc Handling is configurable: Either an internal buffer is used or Crc is stored at the end of RAM Block.” ESCAN00035891: Chapter 7.1.1 “Integrate SWC-Generation into CFG Pro's Generation process” Chapter 3.1: update AUTOSAR architecture figure.
virck	2010-01-25	3.04.01	ESCAN00039648 – Rebuilt document; made hyperlinks working. Updated Logo; No changes in content.
virck	2010-03-26	3.05.00	Updated Component history Whole document: “EAD” “DaVinci Configurator” Added Ch. 7.3 “Attributes only configurable using GCE” Updated Ch. 5.6.1 – “RAM Usage” ESCAN00040662: Chapter 4.4.3: Added note about restricted access to RAM block during job execution. ESCAN00035134: Chapter 5.1.2 reworked ESCAN00039749: Ch. 4.4.10, 8.2.4: Guaranteed CRC values; Ch 6.4.7: note about asynchronous CRC calculation ESCAN00031315: added Ch. 4.2.1, Ch 8.2.3; updated Ch. 7.2.5 ESCAN00042745 – corrected Ch. 4.5.2

Manfred Duschinger	2011-01-25	3.07.00	ESCAN00047171: Ch. 6.4.18: NvM_KillWriteAll; Abbreviations: ECUM ESCAN00045141: Ch. 4.4.5.1: information about names of Block Handles
Manuela Scheufele	2011-02-03	3.07.01	Minor changes
Manfred Duschinger	2011-07-12	3.08.00	ESCAN00049327: Ch. 4.5.2 DEM errors inserted into MICROSAR DEM
virck	2012-01-24	3.09.00	ESCAN00053235, ESCAN00051729: Ch. 7.1 – updated PortInterface names
Manfred Duschinger	2013-01-02	5.00.00	Ch. 4.1: supported features added Ch. 4.4.3 and ch. 4.4.4: added NvM_CancelJobs Ch. 4.3.5: error handling updated Ch. 4.4.20: added explicit synchronization mechanism Ch. 5.4.6: updated callback functions Ch. 5.5: updated initialization process of memory stack Ch. 6.4: updated return values of most synchronous APIs Ch. 6.4.6: instanceID deleted Ch. 6.4.16: updated NvM_WriteAll handling Ch. 6.4.19: description of API NvM_CancelJobs Ch. 6.7.4 and 6.7.5: Callback routines for explicit synchronization mechanism Ch. 7: completely reworked Ch. 9.2: added abbreviations Ch. 7.2.1 and ch. 7.3 removed
Manfred Duschinger	2013-01-04	5.01.00	Ch. 5.4.8 Interaction with BswM
Manfred Duschinger, virck	2013-08-23	5.01.01	ESCAN00064110: Ch. 4.4.8 Description of synchronous Job-End Notification ESCAN00062895: Ch. 4.4.5.1 Symbolic name values of Nv Block Handles updated. ESCAN00062896: Ch. 4.4.13. 4.4.20, 5.4.8, 6.7.3, 6.7.4 and 6.7.5: Added information that block is still busy during invoking callback. ESCAN00063639: Ch. 4.4.20: Extended information about explicit synchronization mechanism ESCAN00064063: Ch. 6.2 Improve description of `NvM_RequestResultType` ESCAN00064173: Ch. 5.3: Explanation of some necessity of memory mapping

			ESCAN00068239: Ch. 6.4, Ch. 4.4.20: Limitations Explicit Synchronization Mechanism ESCAN00063532 – Ch. 6.4.16 Block Processing order during NvM_WriteAll
virck	2014-06-17	5.02.00	Internal release; no changes
virck	2014-10-13	5.02.01	Updated Ch. 2. Component history. ESCAN00073178 – updated Ch. 4.1 unsupported features, Ch. 8.1 Deviations ESCAN00074672 – Description of Redundant Blocks; extended Ch. 4.4.5.2 ESCAN00076366 – SWCs' callback return types – added Ch. 7.1.5 Removed Ch. 4.4.18, 4.4.19 ESCAN00071933 – Description of internal buffering and internal CRC storage, rewording Ch. 4.4.5.1, 4.4.5.5, 4.4.10, 5.6.1 ESCAN00075284 – Reworked Ch. 4.4.17, Ch. 6.4.8 Review findings – Development Error Codes in chapter 4.5.1, minor rephrasing, Glossary (PIM) Added Chapter 5.7
virck	2015-01-07	5.02.02	Chapters 6.4.8, 8.1 NvM_SetBlockLockStatus – emphasized deviation from AUTOSAR.
viroto	2015-02-02	5.03.00	Chapter 2.2.1, 2.5.1 – removed RAM and ROM block length DET check Chapter 2.5.3 – describes the new compile time RAM and ROM block length checks Chapter 2.1.1.1– created to describe feature Block Id check Chapter 4.4.24 – describes the new feature "Repair Redundant Blocks"
viroto	2015-09-28	5.03.01	Only improvements
viroto	2015-11-25	5.0400	Added chapter 2.1.2 and updated chapter 2.5.3
viroto	2016-02-11	5.05.00	Removed the possibility to configure dev error hook, include file and reporting
viroto	2016-04-14	5.06.00	ESCAN00088791: updated function signatures to AUTOSAR4 in chapter 4.4 FEAT-1888: described changed callback invoking during ReadAll in chapters 2.4.8 and 2.4.14 Added new chapter 3.2 with critical section list.

			ESCAN00091004: extended chapter 2.2
viroto	2016-10-18	5.06.01	ESCAN00073639: improved chapter 2.4.19
viroto	2017-02-23	5.06.02	ESCAN00094128: improved DEM error handling description (chapter 2.5.2 and 3.4.2)
viroto	2017-07-19	5.07.00	FEAT-2914: added CRC Compare Mechanism documentation
viroto	2018-05-16	5.07.01	ESCAN00098248: improved chapter 2.4.9
viroto	2018-09-04	5.07.02	ESCAN00099765: extended chapter CRC Compare Mechanism
vircre	2018-10-09	5.08.00	STORYC-5670: Provide an interface to protect and validate NV data (chapters 2.4.15, 4.7.8 and 4.7.9 were added).
virbka	2019-04-03	5.09.00	STORY-8865: NvM shall provide a service to abort an ongoing ReadAll in a save way (chapter 6.4.21 was added).
viroto	2019-04-24	5.09.01	ESCAN00102393 and ESCAN00102760: reworked chapter 2.4.11.
virsrl	2019-07-10	5.10.00	MWDG-47: NvM shall pass the processed block Id to the single block callback. MWDG-102: NvM shall pass the currently pending block Id to the init callback.
viroto	2019-07-15	5.11.00	MWDG-108: NvM shall detect unchanged data and skip writing to NV RAM.
viroto	2019-08-29	5.12.00	MWDG-36: NvM shall encrypt and decrypt data via synchronous CSM interfaces. (chapter 2.4.22)
viroto	2019-09-04	5.12.01	ESCAN00104007: reworked chapters 4.7.2 and 4.7.5.
virsrl	2019-09-12	5.13.00	MWDG-2: NvM shall retry a unsuccessful CSM encrypt and decrypt. (chapters 2.4.23.2.1, 2.4.23.2.2)
virstl, viroto	2019-10-23	6.00.00	MWDG-86: Provide the request result `NVM_REQ_RESTORED_FROM_ROM` in case the default data was loaded during a read request. (Updates chapters 2.4.5.4, 3.7.3, 4.2) Updated chapter 3.
virsrl	2019-11-11	6.00.01	MWDG-1049: Describe necessary configuration in DaVinci Developer for SwComponents of type NvBlockSwComponent.
virstl, virlra	2019-11-20	6.01.00	MWDG-2245: NvM shall support NvMBlockUseSetRamBlockStatus

			configuration. Adds chapter 5.2.1.
viroto, virsrl	2020-01-22	6.01.01	MWDG-2579: Make NvM_Types.h standalone includable (chapter 3.1), ESCAN00105284
virsrl	2020-02-21	6.02.00	MWDG-2303: NvM lets the user configure single block callback invocation during ReadAll. MWDG-2301: NvM lets the user configure single block callback invocation during WriteAll.
virljs	2020-07-20	7.00.00	MWDG-3209: TechRef Improvement Changed wording of NVM_REQ_PENDING in chapter 6.2 MWDG-3318: NvM CRC32 calculation does not match ASR standard
virljs	2020-08-07	7.01.00	MWDG-3657: NvM shall provide a (hidden) backwards compatible CRC32 calculation
virljs	2020-09-30	8.00.00	MWDG-3906: NvM shall use MICROSAR default MemMap section (Chapter 3.3) MWDG-4102: NvM shall recognize loss of redundancy during block reading (Chapter 2.4.22)
viroto	2020-12-17	8.01.00	MWDG-4344: NvM shall support calculate RAM block CRC for block with explicit synchronization (new chapter 2.4.11, reworked chapter 2.4.9)
virlra	2021-03-15	8.02.00	MWDG-4548: NvM tamper detection: Add CSM MAC handling to data integrity service
virsrl	2021-05-04	9.00.00	MWDG-3655: Provide and use the generated NvM_MemMap.h
virsrl	2021-06-25	9.01.00	MWDG-5380: NvM shall provide its JobEnd and JobError notifications compatible to ASR4.4 Fee
virsrl	2021-11-03	9.01.01	MWDG-4779: TechRef Improvements: - Clarifications to NvM_CancelJobs API - Add warning to immediate block usage - Correct redundant block handling

vireno	2021-12-03	10.00.00	Removed CRC32 compatibility mode in chapter 2.4.10. Added additional description for length checks in chapter 2.5.3.
virbka	2022-05-18	10.01.00	“MWDG-4193 : NvM: Split ReadAll into two phases”: Added chapter 2.2.2.1 and chapter 4.4.19.
virbmz	2022-05-25	10.02.00	“MWDG-4194 NvM: Make reading of config block dependent on NvMDynamicConfiguration” Extended chapter 2.2.2 and chapter 4.4.15.
vireno	2022-07-13	10.03.00	“MWDG-7169 NvM: Add separate internal buffer for immediate prio jobs”: Added warning that immediate priority blocks lead to a bigger footprint in chapter 2.4.9.
vireno	2022-09-14	10.04.00	“MWDG-727: NvM: PreReadAll configuration parameter shall not be validated as XOR”: Added chapter 5.2.3 and extended 5.2.4.
virbmz	2022-09-28	11.00.00	“MWDG-7388: NvM AR4.3 service interface compliance”: Added information for new APIs to chapter 2.4, chapter 2.5 and chapter 4.4.
virsrl	2022-11-11	11.01.00	MWDG-7452: Support MAC Compare Mechanism
virsrl	2023-07-27	11.02.00	MWDG-8311: Support block callbacks according to AUTOSAR 4.4.0
smacht	2023-08-17	11.03.00	MWDG-8321: Perform CMSR Component Security Analysis

Table 1-1 History of the document

Reference Documents

No.	Source	Title	Version
[1]	AUTOSAR	Specification of NVRAM Manager	V 3.2.0
[2]	AUTOSAR	Specification of NVRAM Manager	V 4.2.2
[3]	AUTOSAR	Specification of Development Error Tracer	V 2.2.0
[4]	AUTOSAR	Specification of Diagnostics Event Manager	V 2.2.1
[5]	AUTOSAR	List of Basic Software Modules	V 1.2.0
[6]	AUTOSAR	Specification of Crypto Service Manager	V 4.3.0

Table 1-2 Reference documents

Please note
We have configured the programs in accordance with your specifications in the questionnaire. Whereas the programs do support other configurations than the one specified in your questionnaire, Vector´s release of the programs delivered to your company is expressly restricted to the configuration you have specified in the questionnaire.

Contents

Illustrations

Tables

1 Introduction

This document describes the functionality, API and configuration of the AUTOSAR BSW module NVM as specified in [1].

Latest Analyzed AUTOSAR Release:	R19-11	
AUTOSAR Schema Compatibility	R4.0.3	
Supported Configuration Variants:	link-time	
Vendor ID:	NVM_VENDOR_ID	30 decimal (= Vector-Informatik, according to HIS)
Module ID:	NVM_MODULE_ID	20 decimal (according to ref. [5])

The module NVM is created to abstract the usage of non-volatile memory, such as EEPROM or Flash, from application. All access to NV memory is block based. To avoid conflicts and inconsistent data the NVM shall be the only module to access non-volatile memory.

The NVM accesses the module MEMIF (Memory Abstraction Interface) which abstracts the modules FEE (Flash EEPROM Emulation) and EA (EEPROM Abstraction). Thus, the NVM is hardware independent. The modules FEE and EA abstract the access to Flash or EEPROM driver. To select the appropriate device (FEE or EA) the NVM uses a handle that is provided by the MEMIF.

Caution
MICROSAR FEE and MICROSAR EA are different products that are not part of MICROSAR NVM!

1.1 Architecture Overview

The following figure shows where the NVM is located in the AUTOSAR architecture.

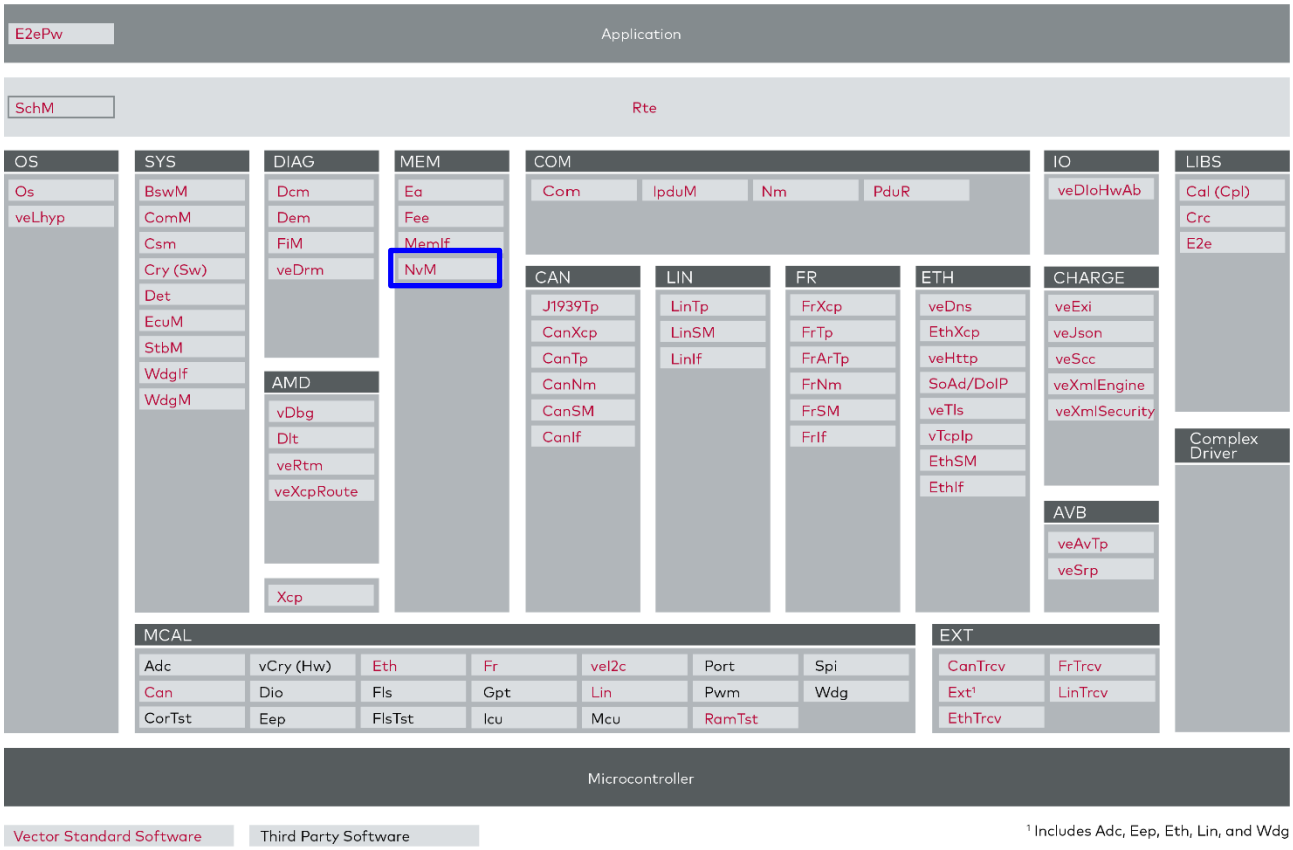

Figure 1-1 AUTOSAR 4.x Architecture Overview

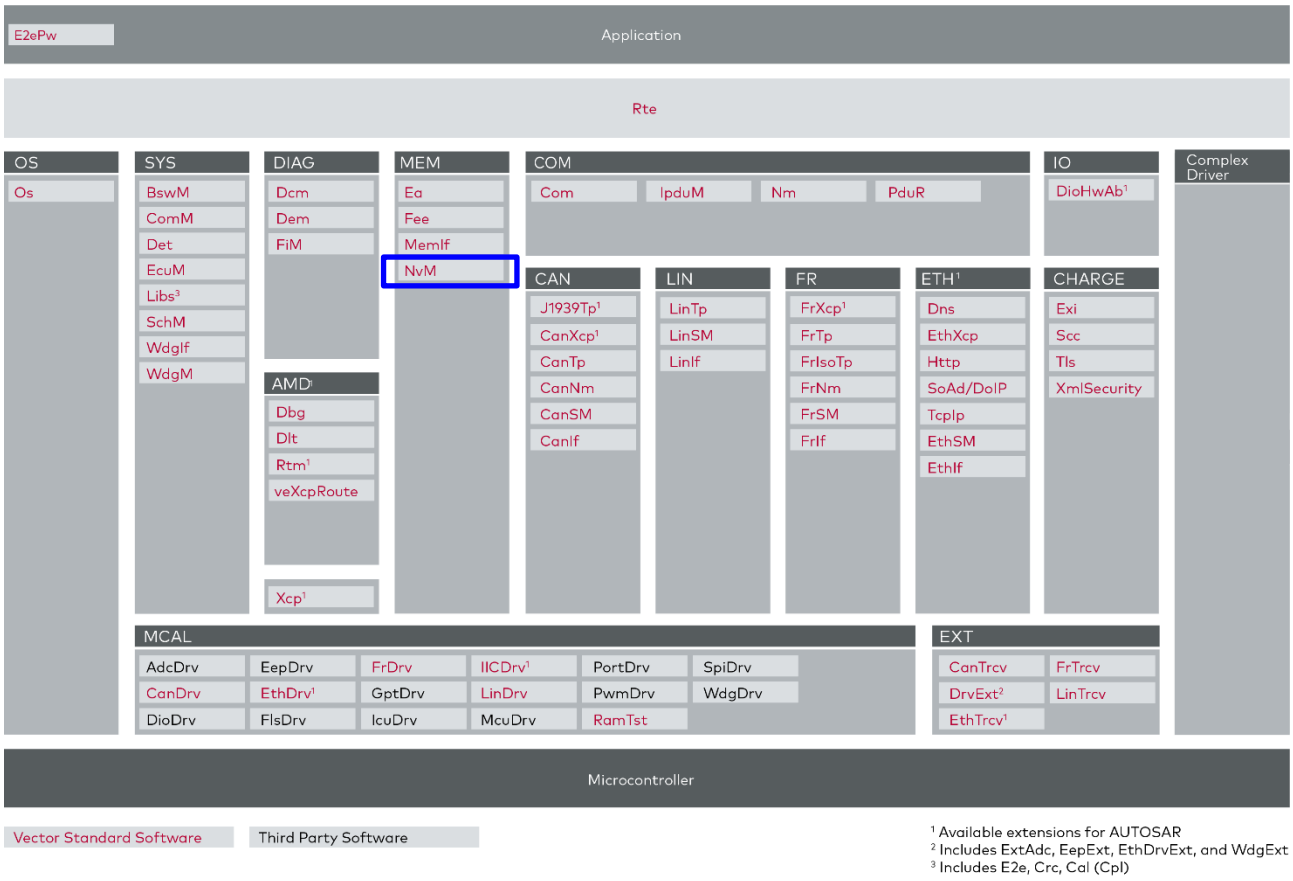

Figure 1-2 AUTOSAR 3.x Architecture Overview

The next figure shows the interfaces to adjacent modules of the NVM. These interfaces are described in chapter 4.

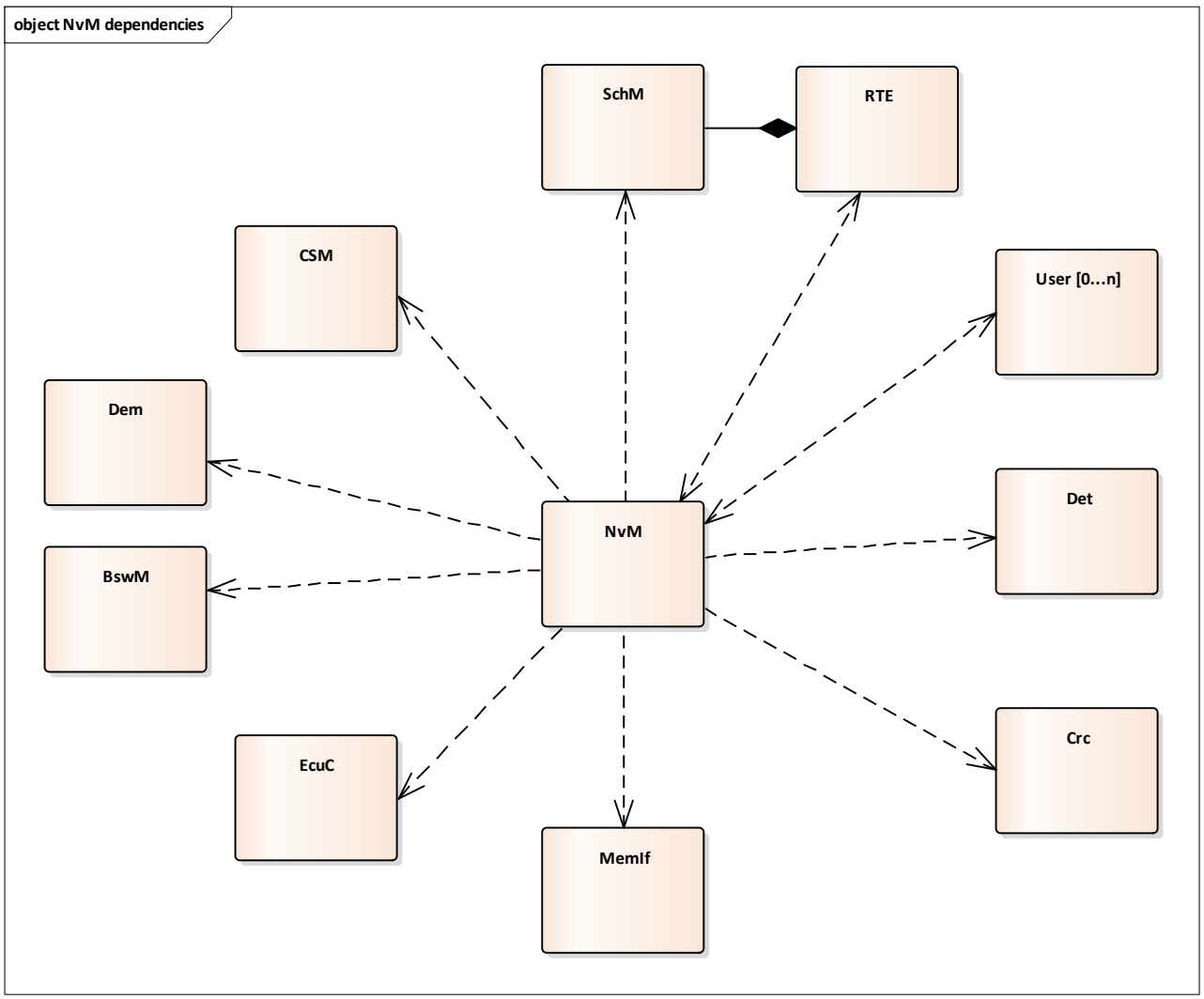

Figure 1-3 Interfaces to adjacent modules of the NVM

Applications normally do not access the services of the BSW modules directly. They use the service ports provided by the BSW modules via the RTE. The service ports provided by the NVM are listed in chapter 4.8 and are defined in [1].

2 Functional Description

2.1 Features

The features listed in this chapter cover the complete functionality specified in [1].

The **supported** and **not supported** features are presented in the following two tables. For further information on not supported features also see chapter 6.

The following features described in [1] are supported:

Supported Feature
Complete API
Block Management Types (Native, Redundant, Dataset)
CRC handling (CRC16, CRC32)
Priority handling, including Immediate (Crash) Data write
Job queuing
ROM defaults (ROM defaults block, Init callback)
Config Id handling
RAM block valid/modified handling
Re-Validation of RAM blocks during start up using CRC
Job end notifications
Skipping Blocks during Start-Up
API Configuration Classes
Service Ports – Generation of Software Component Description
Concurrent access to NV data for DCM
Explicit Synchronization mechanism between application and NVM
Interaction with BswM

Table 2-1 Supported SWS features from main AUTOSAR version [1]

The following features described in [1] are not supported:

Not Supported Feature
Dataset ROM blocks (Management Type Dataset, multiple ROM blocks)
Disabling Set/Get_DataIndex API
Static Block ID Check during read
Write Verification
Read Retries

Table 2-2 Not supported SWS features from main AUTOSAR version [1]

The following features described in [2] are supported

Supported Feature
CRC Compare Mechanism (AUTOSAR deviation: implemented independent from CalcRamBlockCrc)
CalcRamBlockCrc for blocks with Explicit Synchronization Mechanism.

Table 2-3 Supported SWS features from AUTOSAR version [2]

The following features are supported in addition to the AUTOSAR standard:

Additionally Supported Feature
NvMCheckLossOfRedundancy
Use CSM MAC for data integrity checks
Synchronous CSM encryption/decryption
Use CSM MAC to avoid unnecessary write operations
Support of NvM_KillWriteAll
Forward NvM jobs to another core

Table 2-4 Additionally supported features

2.1.1 Safety Features

2.1.1.1 Block Id check

NvM provides a feature to ensure the underlying devices deliver data for the currently processing NvM block – Block Id Check.

Note
Since the Block Id Check is implemented via extension in CRC calculation, the feature is only working for NvM block configured with CRC.

Since the feature uses the NvM Block Id, during configuration update the user has to ensure the Block Id remains the same for each NvM Block. Otherwise the check will fail though the correct data was read.

To check the data to belong to currently processing NvM Block, the NvM calculates the NvM Block Id and the current Dataindex into its CRC. That means in fact that the NvM calculates the CRC over the Block Id (2 bytes), Dataindex (1 byte) and the actual data – NvM needs one CRC calculation function call more that without the Block Id check.

Caution
The NvM is not able to distinguish between wrong CRC and CRC calculated for another NvM Block! In case the underlying modules deliver data belonging to wrong NvM Block, the NvM behaves in same way as in case of CRC mismatch.

2.1.2 Automatic Block Length

Since the block length might be unknown during configuration time, the feature Automatic Block Length can be enabled for NvM blocks with permanently configured RAM.

The feature changes the meaning of block length from actual length to maximum length - the actual block length is set via size of permanent RAM within generated structures. The configured length is used by underlying modules to initialize their structures, therefore it must not be less than the actual length. To check the configured length to be valid, Block Length Check (see 2.5.3) shall be enabled.

Caution
After the system is running and the actual block lengths are known, the configuration shall be adjusted to the actual length. Since the configured lengths are used by underlying modules, there might be a lot of unused space in Flash or EEPROM.

2.2 Initialization

Before the module NVM can be used it has to be initialized. All modules, NVM depends on, need to be initialized before. The initialization of all these modules should be done by the ECU State Manager. If the NVM is not used in an AUTOSAR environment it should be done by a different entity. Pay attention that the NVM **will not** initialize the used modules by its own.

Depending on the configuration of the NVM stack, different modules might need to be initialized. It is advised to use a bottom up strategy for initialization:

- NV device drivers for internal devices (FLS/EEP)
- Low level driver that an external NV device driver might depend on (e.g. DIO, SPI)
- Drivers for external NV devices (e.g. external EEP or FLS)
- NV device abstraction modules (EA/FEE)
- Non-Volatile Manager (NVM)

Initializing the modules in this sequence ensures that, as soon as a module is used, the modules it depends on are ready.

Basic Knowledge
NvM initialization consists of two steps

1. `NvM_Init()` (see 2.2.1)
2. `NvM_ReadAll()` (see 2.2.2)

Indenpendently from `SelectBlockForReadAll` NvM uses the `NvM_ReadAll()` to initialize all its blocks. Therefore it is not possible to access any NvM block until it was initialized during `NvM_ReadAll()`.
There is also the optional possibility to split the ReadAll into two phases (see 2.2.2.1)

2.2.1 Start-up

The basic initialization of the NVRAM Manager is done by the request `NvM_Init()`. It shall be invoked e.g. by the ECU State Manager exclusively. Due to strong constraints concerning the ECU start-up time the `NvM_Init()` request does not contain the basic initialization of the configured NVRAM blocks. The `NvM_Init()` request resets the internal variables of the NVM such as the queue and the state machine.

2.2.2 Initialization of the Data Blocks

The initialization of the single blocks is normally also initiated by the ECU State Manager by calling `NvM_ReadAll()`. All blocks that have no valid RAM data anymore and have `SelectBlockForReadAll` set will be reloaded from NV memory or from ROM (if available). All other blocks won't be reloaded, they must be loaded manually by the application calling `NvM_ReadBlock()`, but they will be initialized, e.g. their write protection and status.

NvM's configuration block (block 1) is a pre-configured redundant NvM block which always shall exist. NvM stores the current configuration ID in it. This block is stored in NV memory and also as a constant (`NvM_CompiledConfigId_t`) that is externally visible and link-time configurable.

The configuration block is used to recognize configuration updates (new configuration ID) during start up (`NvM_ReadAll()`). To determine an update the NV value of block 1 is compared against the constant `NvM_CompiledConfigId_t`.

In case **Dynamic Configuration** is **disabled**:

The configuration block shall be ignored, therefore it will not be read out at all and a mismatch is ignored. The NvM processes the normal runtime preparation for all blocks. That means all blocks are handled normally.

In case **Dynamic Configuration** is **enabled**:

The configuration block is read and checked for mismatch.

- In case the **configuration ID matches** the current one, the NvM processes the normal runtime preparation for all blocks. That means all blocks are handled normally.
- In case the **configuration ID does not match** the current one, block could not be read, or the CRC does not match the recalculated one, the NvM behaves following:
  - The configuration block is updated and prepared to be rewritten during next multi block write request.
  - Blocks which are resistant to changed SW receive a normal runtime preparation.
  - Blocks which are not resistant to changed SW receive an extended runtime preparation. All blocks that will be processed with the ‘extended runtime preparation’ will be treated as invalid or blank. Thus, it is possible to rewrite a block having been marked as ‘Write Once’. If available, ROM defaults are loaded or the initialization callback is invoked.

2.2.2.1 Splitting initialization of the Data Blocks into two phases

The overall ReadAll processing can be optionally splitted into two phases. Before actually calling asynchronous `NvM_ReadAll()`, the asynchronous `NvM_PreReadAll()` service can be called and processed.

The multi block job `NvM_PreReadAll()` reads all NvM blocks configured with enabled `NvMSelectBlockForPreReadAll` from NV RAM. When this service is executed, the data of the blocks configured to be read at this first phase will be loaded from NV RAM to corresponding permanent RAM and will be validated. Afterwards, NvM is operable for blocks already read out.

The blocks already processed within the `NvM_PreReadAll()` will be ignored within the `NvM_ReadAll()` later. A block can be either read out in the PreReadAll or in ReadAll phase, not in both.

Note

- `NvM_PreReadAll()` has the same behavior as `NvM_ReadAll()` regarding callback handling. Therefore, it will also use (if configured) Multi Block, Single Block and BswM Status Information Callbacks.
- `NvM_PreReadAll()` will NOT read out the ConfigBlock, therefore functionality in context with “dynamic Configuration”, “extended Runtime Preparation” and “Resistance To Change Software” are not part of the PreReadAll processing.
- `NvM_PreReadAll()` can NOT be killed via KillReadAll service
- `NvM_PreReadAll()` will NOT check the CRC of RAM blocks with CRC in advance. Therefore, selected blocks will be always read out even if the block may be still valid in RAM.

2.3 States

The NVRAM Manager is internally organized with a state machine which is shown in the following chapters.

2.4 Main Functions

2.4.1 Hardware Independence

The NVRAM Manager is independent from its underlying memory hardware. It accesses the API of the MEMIF (Memory Abstraction Interface). The MEMIF abstracts the modules FEE (Flash EEPROM Emulation) and EA (EEPROM Abstraction) for the NVM. FEE and EA are used for storing data blocks in Flash or EEPROM devices. For selecting at which FEE or EA instance a block shall be stored, the NVM uses a device handle (device ID) that is provided by the MEMIF.

2.4.2 Synchronous Requests

The NVM API services are divided into synchronous and asynchronous requests.

The synchronous services are executed immediately when called. They are executed in the context of the calling task. This means, that behavior depends on the characteristics of the calling task and not on the NVM. For example, if the calling task is a non-preemptive one, the synchronous NVM request will be executed until it has finished. Otherwise, if the calling task is a preemptive one, the synchronous NVM request can be preempted by another higher prioritized task.

Following NVM API services initiate synchronous requests:

- `NvM_Init()`
- `NvM_SetDataIndex()`
- `NvM_GetDataIndex()`

- `NvM_SetBlockProtection()`
- `NvM_SetBlockLockStatus()`
- `NvM_SetRamBlockStatus()` (for not CRC protected blocks)
- `NvM_GetErrorStatus()`
- `NvM_GetVersionInfo()`
- `NvM_CancelJobs()`
- `NvM_CancelWriteAll()`
- `NvM_KillWriteAll()`
- `NvM_KillReadAll()`

2.4.3 Asynchronous Requests

Following NVM API services initiate asynchronous requests:

- `NvM_ReadBlock()`
- `NvM_ReadPRAMBlock()`
- `NvM_WriteBlock()`
- `NvM_WritePRAMBlock()`
- `NvM_RestoreBlockDefaults()`
- `NvM_RestorePRAMBlockDefaults()`
- `NvM_EraseNvBlock()`
- `NvM_InvalidateNvBlock()`
- `NvM_SetRamBlockStatus()` (for CRC protected blocks)
- `NvM_ReadAll()`
- `NvM_PreReadAll()`
- `NvM_WriteAll()`
- `NvM_RepairRedundantBlocks()`

The API call is handled in the context of the calling task. Here the service is queued and will be processed asynchronously. The processing of the queued requests is done in the context of the caller of the cyclic function `NvM_MainFunction()`.

Caution
RAM blocks must not be accessed by any user while a request to its associated NVRAM Block is pending!
There are some exceptions to this limitation:

- `NvM_InvalidateNvBlock` and `NvM_EraseNvBlock` don't access any RAM blocks. Thus access is still possible without limitations
- While the NVM processes an `NvM_WriteBlock` request, the RAM block may still read.
- Though applications are not expected to be running while NVM processes `NvM_WriteAll`, RAM blocks may be read, as during `NvM_WriteBlock` processing.

2.4.4 API Configuration Classes and additional API Services

Depending on the needs of the customer, the extent of the NVM can be tailored. Three configuration classes are specified that offer a different amount of functionality/functions of the NVM:

API configuration class 1:

A minimum set of API services is used. Queuing and job prioritization are not implemented. Following functions are available:

- `NvM_Init()`
- `NvM_SetDataIndex()`
- `NvM_GetDataIndex()`
- `NvM_GetErrorStatus()`
- `NvM_ReadAll()`
- `NvM_KillReadAll()`
- `NvM_WriteAll()`
- `NvM_CancelWriteAll()`
- `NvM_MainFunction()`

API configuration class 2:

Intermediate set of API services. Queuing and job prioritization are implemented. Following functions are available additionally according to API configuration class 1:

- `NvM_ReadBlock()`
- `NvM_ReadPRAMBlock()`
- `NvM_WriteBlock()`
- `NvM_WritePRAMBlock()`

- NvM_RestoreBlockDefaults()
- NvM_RestorePRAMBlockDefaults()
- NvM_CancelJobs()

API configuration class 3:

All API services are available. Following functions can be used additionally to API configuration class 2:

- NvM_SetBlockProtection()
- NvM_EraseNvBlock()
- NvM_InvalidateNvBlock()

The functions NvM_SetRamBlockStatus(), NvM_PreReadAll(), NvM_KillWriteAll(), NvM_RepairRedundantBlocks and NvM_GetVersionInfo() can be enabled/disabled additionally via the configuration tool.

The function NvM_SetBlockLockStatus() is always available independent of API configuration class.

2.4.5 Block Handling

2.4.5.1 NV Blocks and Block Handles

Every application's data packet that is intended for storage in NV memory is seen as a block. For each block a unique block handle (block ID) is used. For the application the (RAM) block is just one of its variables associated with the block. To write this variable to NV memory it calls the NvM_WriteBlock() service with the block handle that is mapped to this variable. The block handle names are given during configuration of the NVM. They are published to the application by including NvM.h.

Note
The block handle names are automatically prefixed according to the AUTOSAR Specification EcucConfiguration:
<Module Definition>Conf_<Container Definition Short Name>_<Container Instance Short Name>
The prefixing has no influence on RTE.

Caution
The actual processing of an asynchronous job (such as a write job) is done in `NvM_MainFunction`. Therefore it needs to be called cyclically. Usually this is done by the Basic Software Scheduler (SCHM).

2.4.5.2 Different Types of NV Blocks

The application data can be stored in different types of blocks in the NV memory.

2.4.5.2.1 Native Blocks

This is the standard block type. The data is stored once in the NV area.

2.4.5.2.2 Redundant Blocks

This type is intended to increase **availability** of data, in case of errors, i.e. it is not intended to provide additional error detection. The main focus lies on write aborts, especially resets due to under-voltage conditions.

Note
It is recommended to configure a data integrity mechanism (e.g. CRC or MAC) usage for Redundant Blocks, because they provide adequate **error detection**, beyond the scope of aborts.

The user data is stored twice in the NV area. While relying on lower layers' (FEE/EA) detection of aborted write accesses, NVM makes sure that a readable data block remains readable, even in case of write aborts. For that purpose, before starting a write access, NvM checks primary and secondary NV blocks to determine the adequate write order (which NV block to write first). If a defective NV block is detected, the NvM will attempt to write this block first.

Error handling of redundant blocks:

- If writing to both NV blocks failed, the NvM reports the error NVM_E_REQ_FAILED (see chapter 2.5.2) to DEM.
  Result of the write job will be NVM_REQ_NOT_OK.
- If writing to only one of the NV blocks failed, the NvM reports the error NVM_E_LOSS_OF_REDUNDANCY (see chapter 2.5.2) to DEM.
  Result of the write job will be NVM_REQ_OK.

A read request is successful even if one block is corrupted but the other block could be read. An erase or invalidate request is only successful if both blocks could be erased respectively invalidated.

Expert Knowledge
After a write abort, the “age” of data is not defined. NVM may deliver previous or recent data; in fact it does not distinguish them. Before NVM completed the result with NVM_REQ_OK, clients shall make no assumption on “age” of data in NV memory.

Expert Knowledge
NVM does not check any data to determine the write order. Rather, it just checks whether lower layers would find valid data instances (i.e. whether they successfully read a block’s first data byte). At this point, NVM relies on lower layers’ abort detection capabilities.

Note
NVM always attempts writing both NV blocks, regardless of errors reported by lower layers.

Note
An additional feature was introduced for redundant NV blocks to detect loss of redundancy, see 2.4.22

2.4.5.2.3 Dataset Blocks

A dataset block can be seen as an array. A configurable number of instances of this block are stored in NV-memory. In the RAM area there is only one RAM buffer. The appropriate NV block instance is selected by the so called **data index**. The data index can be read and set by synchronous API services `NvM_GetDataIndex()` and `NVM_SetDataIndex()`.

Concept	Description
Block	General notion of the structure composed of data, state and CRC. It is spread over RAM, ROM and NVRAM.
NV Block	One block in NVRAM – CRC is optional.
NV Block of > Native type > Redundant type > Dataset type	One NV Block of specified type.
RAM Block	One data Block in RAM. The data is shared by NVRAM Manager and application. E. g. application writes data to this block and requests NVRAM Manager to write it into NVRAM.
ROM Block	One data block in ROM. Default data supplied by application.
NVRAM Block	A logical composition of one RAM block and its corresponding NV and ROM Block.
NV = NVRAM	Non-volatile memory. Actually a synonym for Flash or EEPROM devices.

Table 2-5 Block concept

2.4.5.3 Permanent and non-permanent RAM Blocks

The RAM block (application variable) can be either permanent or non-permanent. A permanent RAM block belongs to a NV block that is accessed only by one application. The address of the RAM block is fixed and is stored in the configuration of the NVM.

It is also possible to have multiple applications accessing the same NV block. Each application uses its own RAM block. In this case the RAM block is called non-permanent. As the RAM address is not stored (and may vary) a pointer must be given for reading and writing a non-permanent block.

Caution
Asynchronous API functions can be reentered by different tasks. So it is possible that several tasks queue for example a write job at the same time (a task with higher priority might interrupt a lower one). But it is not possible to queue the same block multiple times (neither by different tasks nor for different jobs). So if for instance a read job for block 5 is queued, an erase job for this block can't be queued before the read job is finished.

If one block is used by multiple tasks, which is a common task for non-permanent RAM blocks, the application is responsible for synchronization. Of course if, for example, an erase request is in process the RAM block could be read or written without any effect to the result of the erase job. The only problem is that the NVM does not offer any information to an application what service is currently processed for a block. The application that initiated the service of course does know, but a different application that also uses the block does not. So the safest way for block access is not to use the RAM block as long as it is **pending**. This way RAM inconsistency can be avoided definitively.

2.4.5.4 ROM Defaults

ROM defaults can be assigned to any NVRAM block. The ROM defaults block is provided by the application. Alternatively, an initialization callback (see 2.4.14) can be used. These features are selected during configuration. It is only possible to configure either ROM defaults or an initialization callback for a block.

ROM defaults can be read explicitly (by a call of `NvM_RestoreBlockDefaults()`). ROM defaults will also be read implicitly during a read request, if no valid data could be read from NV-memory, either due to a data integrity validation error or because of a failure reported by the underlying MemHwA via MEMIF.

In case ROM default data is loaded during read operation of the block, the result will be `NVM_REQ_RESTORED_FROM_ROM`. A call to `NvM_RestoreBlockDefaults()/RestorePRAMBlockDefaults()` will not provide the result `NVM_REQ_RESTORED_FROM_ROM`.

2.4.5.5 Data Integrity

The NvM provides multiple strategies to ensure the data integrity of a configured block, i.e., it can check if the block data read from NV-memory corresponds to the data that was previously written.

Note
Only a single data integrity mechanism can be configured for a block at a time. Additional features that require a dedicated strategy may not be available if a different data integrity mechanism is configured.

If `NvMDataIntegrityIntBuffer` is disabled, storage for data integrity values (e.g. CRC or MAC) must be provided by every single user; otherwise NVM provides an internal buffer. In this case it copies user data (associated with NVRAM blocks configured with integrity value) into an internal buffer, instead of directly passing them down to lower layers. Here,

data gets appended with the integrity value, in order to keep both within one NV block, which requires NVM to pass both down with one single write request.

2.4.5.5.1 CRC

For each block an optional CRC checksum can be configured. This checksum can be either CRC16 or CRC32. The checksum will be appended to user data; in NV memory they will be stored consecutively in one single NV block.

If `NvMCalcRamBlockCrc` was additionally enabled, NVM will internally store CRC values in RAM, in order to check against them during `NvM_ReadAll` processing. Without internal buffering, additional space in RAM block serves for this purpose.

The CRC support is an AUTOSAR feature.

2.4.5.5.2 MAC

For each block an optional MAC can be configured. Because the NvM doesn’t implement any MAC functionality, the CSM module is used to generate and verify MACs. The generated MAC will be appended to the user data and both are stored in a single block in NV-memory.

The MAC support is a non AUTOSAR feature.

More information on how to configure MAC for a block can be found at 5.2.4.

Note
When configuring MAC for a block, some features become unavailable that require a different data integrity mechanism to be used (e.g. `NvMCalcRamBlockCrc` or `NvMUseBlockIdCheck` who require CRC).

2.4.6 Prioritized or non-prioritized Queuing of asynchronous Requests

As mentioned before, asynchronous services are not processed immediately but queued and processed asynchronously by the `NvM_MainFunction()`. This is necessary to decrease the runtime of application tasks and to increase the predictability of their duration (synchronous write jobs on an EEPROM or Flash would block your task for multiple milliseconds up to one second).

Jobs can be queued either prioritized or non-prioritized, depending on the user configuration.

If job prioritization is configured, the priorities 0 (immediate priority) until 255 (lowest priority) can be selected for a block. It is important that the priority depends on the block, rather than the request. Multi block requests always have a priority value greater than 255, i.e. their priority is less than the lowest block specific priority; they will be processed after all single block requests have been completed.

If block prioritization is not selected, the job queue works as a FIFO buffer.

2.4.7 Asynchronous Job-End Polling

As alluded before, asynchronous requests are processed in the background. The application has the possibility to poll the NVM for the end of the service by calling `NvM_GetErrorStatus()`. `NVM_REQ_PENDING` will be returned as long as the job is queued or in process. Once the job is finished `NvM_GetErrorStatus()` will return the job result.

2.4.8 Single Block Job End Notifications

Alternatively to poll for the job-end, a job end notification can be implemented and configured for every block. NvM invokes the notification each time a job was processed for the block and informs about the finished job and its result via parameter.

Note
The return value of the functions is specified but will not be used by the NVM.

2.4.9 Immediate Priority Jobs and cancellation of current Jobs

Normal priority blocks, priority [1,255], do not cancel jobs with lower priority, NvM will wait until the job is done and then pop the next job with highest priority from its queue.

Only the NvM blocks with immediate priority, 0, must be processed as soon as possible. Depending on current processing state following situations are possible:

- NvM does not process any job: process the immediate priority job.
- NvM processes a job: suspend the job, process the immediate priority job and finally re-start the suspended job.
- NvM waits for underlying module to process a job: cancel the underlying module, suspend the job, process the immediate priority job and finally re-start the suspended job.

Before the suspended job can be re-started, all immediate priority jobs will be processed.

NvM behaves the same for single and multi block jobs, i.e. a multi block job will also be suspended and re-started after the immediate priority job is done.

Caution
Writing to an immediate block usually results in a call to MemIf_Cancel() (when another job with lower priority was already running). Cancellation on lower layer can be time consuming depending on hardware or implementation.

Using many blocks with immediate priority can lead to a higher frequency of write requests on hardware layer.

Note: Immediate writing to blocks is meant for crash data or other exceptional events.

Caution
Some FEE implementations do not support the AUTOSAR defined sequence of:
- Cancel an ongoing job
- Accept a new job after cancellation API has finished

This will cause the immediate priority write handling to fail (and raise DET if enabled) and therefore should not be used in that case.

Some FEE implementations on the other hand will support the above sequence but will wait for the ongoing job to finish and queue the newly requested one. In this case the immediate priority writing will be compromised in sense of runtime.

Please check the feature and error list of your FEE supplier before integrating this feature.

Caution
Pay attention that only blocks with high priority (0) can be erased (by using API `NvM_EraseNvBlock`)!

Note

The usage of immediate priority blocks increases the footprint if internal buffering is enabled. This is the case when one of the following parameters is enabled

- `NvM/NvMCommonVendorParams/NvMJobForwardingToMemoryCore`
- `NvM/NvMCommonVendorParams/NvMDataIntegrityIntBuffer`
- `NvM/NvMCommon/NvMRepairRedundantBlocksApi`

In case of an ongoing `NvM_ReadBlock()/NvM_ReadPRAMBlock()` it cannot be ensured that neither an underlying component nor the hardware is still using the internal buffer from `NvM` after cancellation of the job. To avoid data corruption due to reuse of the internal buffer, an additional internal buffer for the immediate write job is necessary.

2.4.10 Asynchronous CRC Calculation

The (re-)calculation of a block's CRC is done asynchronously by the `NvM_MainFunction()`. To be able to split a CRC calculation job, the number of CRC bytes to be calculated during one cycle can be configured via the configuration tool.

A CRC protected block's CRC value is calculated

- And added to the data before the block data is written to the NV RAM.
- After reading of the block data to verify its integrity (recalculated CRC must match the stored one).

Note
If an AUTOSAR compliant CRC library implementation is used, the NVM ensures for all supported CRC types that calculated values do not depend on the number of cycles needed for calculation, i.e. for any number of calculation steps any CRC value is guaranteed to be equal to the CRC value calculated over same data with one single call to the appropriate library function.

2.4.11 Calculate RAM block CRC

Enabling this feature for a NvM block leads to an asynchronous CRC recalculation during the MainFunction (in background) each time `NvM_SetRamBlockStatus()` is called for the block with `BlockChanged` value `TRUE`.

The purpose of requesting recalculation of the RAM CRC with every call to `NvM_SetRamBlockStatus()` is to provide the possibility to re-use the RAM data even if a reset (short power-loss, watchdog-reset) occurred. NvM assumes the data in RAM is newer than the data in NV RAM.
NvM attempts so during `NvM_ReadAll` processing for all NVRAM blocks having `SelectBlockForReadAll` and `CalcRamBlockCrc` enabled in their configuration: If the block is internally still marked as VALID, NVM calculates the CRC value over current RAM block's contents and compares it with the value stored elsewhere. If they match it does not touch RAM contents; rather NVM pretends having successfully read those values from NV.

The Calculate RAM block CRC feature is available for block configured either with a permanent RAM block or with Explicit Synchronization Mechanism:

- Permanent RAM holds the latest data to calculate the CRC for
- The configured `NvMWriteRamBlockToNvCallback` must provide the latest data to calculate the CRC for.

Note
In case a NvM block configured to use the Explicit Synchronization Mechanisms uses the Calculate RAM block CRC feature, the NvM will use an internal buffer to get the data from the user via the configured `NvMWriteRamBlockToNvCallback`.

This happens through the block is idle/ not requested!

Since the NvM calculates the RAM block CRC parallel to job processing, there will be two internal buffers!

> One for the job processing (will also be used by Explicit Synchronization Mechanism)

> One for the Calculate RAM block CRC feature.

2.4.12 Write Protection

2.4.12.1 Modifiable write protection

The NVM supports write protection of any NV Block. The API services `NvM_SetBlockProtection()` is used for protecting and unprotecting a NV block. The initial write protection (after reset) can be configured. It will be set during `ReadAll` request.

2.4.12.2 Write once protection

Some data shall be stored only once, overwriting shall be forbidden. To achieve this NvM provides the write once configuration parameter for each NvM block. The actual protection is not visible to the user, it cannot be queried, but the NvM will reject all write/ erase and invalidate requests to such a block.

NvM sets the protection during the `NvM_ReadAll` or `NvM_ReadBlock`/`NvM_ReadPRAMBlock` request as follows:

> Block readable, data available and valid: the block was already written and is write protected.

> Block not readable or data invalid (data integrity validation error): no valid data available, the block can be written.

To initialize the protection of write once blocks, the NvM will read all of this blocks whether they are selected for a ReadAll or not. Behavior during ReadAll based on configuration:

> Write once block selected for ReadAll:

  Reads all write once blocks selected for ReadAll normally and sets the write protection. Error status matches the read result.

> Write once block **not** selected for ReadAll:

Reads a byte of the data for blocks not selected for ReadAll, just to initialize the write protection according to readability. Error status is `NVM_REQ_BLOCK_SKIPPED`, because no data will be provided.

Caution
For all blocks with write once protection that are not selected for ReadAll, the NvM reads only one single byte. Therefore, even if a data integrity mechanism is configured the integrity value will not be verified and the block will be marked as write protected although the data may be invalid.

In case a write once block is configured to use a data integrity mechanism, it shall be selected for ReadAll.

After writing a not protected write once block successfully, NvM sets the block to protected.

The write protection of a write once block cannot be removed by an API call. Nevertheless, it is possible to rewrite such a block by using the extended runtime preparation during `NvM_ReadAll()` (see [2]).

Caution
Pay attention, for a dataset block configured as write once only one dataset can be written. The other datasets can't be written any more. The whole block is protected after first write.

Caution
After a `NvM_KillReadAll` request NvM will not access the NV RAM any more. Therefore, the NvM cannot validate the NV RAM content and set the write protection of a write once blocks – NvM will skip the blocks and ensure they are not write protected!

The block owner must read the block via `NvM_ReadBlock` to allow the NvM to set the write protection according to the NV RAM content.

2.4.13 Erase and Invalidate

There are two services specified for making a NV block unreadable: `NvM_EraseNvBlock()` and `NvM_InvalidateNvBlock()`.

Invalidating a block is much faster than erasing the block because only the status information will be invalidated.

2.4.14 Init Block Callbacks

For any block ROM defaults (see 2.4.5.4) or an initialization callback can be configured. The initialization callback is called every time the default values of the block have to be loaded, e.g. during a restore block defaults service or for failed read jobs.

In contrast to ROM defaults NvM does not update the RAM data itself, this shall be done within the initialization callback.

The return value of the functions is specified but will not be used by the NVM.

Note

> Init callback is invoked when the related block is still busy, so no request shall be issued until the block isn't busy any more.

> If the initialization callback is called for a specific block during `NvM_ReadAll()` is dependent on the block paramater `NvMInvokeCallbacksForReadAll`.

2.4.15 Protect and validate NV data

In some cases, the user may want to transform the data before writing it to or after reading it from NV RAM: e.g. the user may want to implement his own CRC or MAC, encrypt or decrypt and verify the data or implement an old data detection.

In this case the user can configure two data transformation callbacks:

> pre-write transformation: called directly before writing the data to NV RAM. NvM will then write the transformed data to NV RAM.

> post-read transformation: called after reading the data from NV RAM. NvM will forward the transformed data to the user. In case the return value of this function is E_NOT_OK, the NvM will consider the read job as unsuccessful.

Caution

In both cases the user is responsible for correct data length. The transformation callback must not change the configured data length. If additional data shall be stored, e.g. a CRC value, the user has to configure the NvM block length accordingly and provide a buffer which is large enough to store all the required data.

Note

> The post-read transformation is not always possible! If the read job fails, the callback will not be invoked.

> The transformations are configurable pair-wise only: either both callbacks are configured or neither of them.

2.4.16 Define Locking/ Unlocking Services

In preemptive systems, it is necessary to protect some actions of preemption. That means that a few NVM internal actions need to be atomic. So for protecting these sequences functions for entering and leaving such a critical section can be configured. By default the Operating System (OS) services are used.

The configuration tool can be used to define or configure services such as the OSEK services `GetResource(…)` and `ReleaseResource(…)` to lock and unlock resources. To use these services of your Operating System, you must also publish the header file of the Operating System via configuration tool (in the ‘MyECU’ window and the included tab ‘OS Services’).

2.4.17 Interrupts

When interrupts occur during write accesses, they do not corrupt already saved data or data to be written. To ensure this, these critical sections have to be locked, which is configurable via configuration tool.

2.4.18 Data Corruption

Write operations to non-volatile memories are non-atomic operations. A power supply failure during write accesses may lead to corrupted/invalid data. Assuring that corrupted data will not be signaled as valid is no more the task of the NVM but of the FEE or EA.

2.4.19 Concurrent access to NV data for DCM

NVM provides possibility to access NV data concurrently with NVM’s applications. Therefore each configured NVRAM block has an additional alias, the “DCM block”.

Aliases have following differences to normal NvM blocks:

- Aliases have the same configuration as the origin NvM blocks (e.g CRC or length)
- Aliases are treated as NVRAM blocks without permanent RAM block
  - Aliases are neither read at start-up (during `NvM_ReadAll` processing) nor written at shut-down (during `NvM_WriteAll` processing)
  - explicit read or write requests must supply a reference to a temporary RAM block
  - `NvM_SetRamBlockStatus` invoked for an alias does not have any influence on processing
- Only one asynchronous request for an alias can be queued at a time
  - If one is already queued, the request will be rejected (API returns `E_NOT_OK`)
  - `NvM_GetErrorStatus` works for all aliases, no matter which alias ID is given to the function.
    - There is only one job result for all aliases which is valid until the next alias is requested
  - `NvM_SetDataIndex` and `NvM_GetDataIndex` work for all aliases, no matter which alias ID is given to the functions
  - `NvM_SetBlockProtection` works for all aliases, no matter which alias ID is given to the function

All jobs of DCM are always put into **Standard Job Queue**, even if blocks with immediate priority are requested and job prioritization was enabled. So cancellation of pending jobs by an immediate DCM-Block is avoided. The original priority itself is kept.

For accessing the alias of a NVRAM block, NVM provides the global macro `NvM_GetDcmBlockId(<NvMBlockId>)` which expects the origin NVRAM BlockId as parameter and returns the block's alias of type `NvM_BlockIdType`.

Note
It is recommended that DCM accesses NVRAM data only via aliases. Otherwise the DCM would be responsible for synchronization with every single NVM client (blocks' owners).

Caution
DCM should lock the block using `NvM_SetBlockLockStatus` (see chapter 4.4.8) before requesting jobs (via the alias, especially write requests). In case of an error during job processing, DCM should also unlock the block again. If job processing completes successfully the block should remain locked; it will be automatically unlocked after next start-up (`ReadAll` processing).

A lock itself only affects the original block (i.e. the alias cannot be locked).

2.4.20 Explicit synchronization mechanism between application and NVM

NvM supports an optional explicit synchronization mechanism between application and NvM. 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.

The synchronization mechanism can be configured for every NVRAM block separately. If the synchronization mechanism is configured NvM uses the internal buffer as RAM mirror between NvM and application. It is the same internal buffer which is used for data integrity calculation (see chapter 4.4.5.1). The size of the internal buffer is the size of the biggest configured block plus the size of the configured data integrity record (e.g. CRC or MAC).

If the synchronization mechanism is configured, both NvMWriteRamBlockToNvM and NvMReadRamBlockFromNvM must be configured.

It is not useful to configure a permanent RAM block for a block which uses the synchronization mechanism. In this case the RAM block will be ignored. It is also not recommended to configure an Init callback for a block using synchronization mechanism.

Note
If Explicit Synchronization was configured for a block, clients may modify RAM contents (which are not visible to NVM) while block is pending. In this case take care they may get overwritten when a pending read completes.

Basic Knowledge
By definition, this mechanism serves as permanent RAM block.

Expert Knowledge
Calculate RAM CRC and related fast re-validation of RAM data during `NvM_ReadAll` processing cannot be used along with explicit synchronization mechanism.

2.4.20.1 Explicit synchronization mechanism during write requests

After application issued `NvM_WriteBlock/NvM_WritePRAMBlock`, application might modify the RAM block until callback NvMWriteRamBlockToNvM is called by NvM. If NvMWriteRamBlockToNvM is called, application has to provide a consistent copy of the RAM block to the internal RAM mirror.

Note
During calling NvMWriteRamBlockToNvM callback related block is still busy. No request for it shall be issued as long as block is busy.

2.4.20.2 Explicit synchronization mechanism during read requests

After application issued `NvM_ReadBlock/NvM_ReadPRAMBlock`, application might modify the RAM block until the routine NvMReadRamBlockFromNvM is called by the NvM. If NvMReadRamBlockFromNvM is called, then application has to copy the data from the internal RAM mirror to the RAM block.

Note
During calling NvMReadRamBlockFromNvM callback related block is still busy. No request for it shall be issued as long as block is busy.

2.4.21 Avoid unnecessary write operations

In order to avoid unnecessary write operations in NV memory, (i.e. the data of a specific RAM block was not updated during runtime) the NvM module offers a compare mechanism which can be applied while processing a write job.

This compare mechanism is based on the configured DataIntegrity mechanism (see Data Integrity). Before writing data to NV memory NvM recalculates the DataIntegrityRecord for current data and compares it to the previously read or written DataIntegrityRecord:

- If the DataIntegrityRecord matches, NvM assumes the data has not been changed (i.e. is up to date with NV memory) and won't write but finishes the job successfully.
- If the DataIntegrityRecord does not match, NvM assumes the data has been changed (i.e. is not up to date with NV memory) and will write the data to NV RAM.

In some cases, NvM will write even though the RAM content did not change (DataIntegrityRecord match):

- Previous read has failed
- Previous write has failed
- Previous write has led to a loss of redundancy (see also chapter 2.4.22)
- Previously the block was invalidated or erased
- Previously the default data was restored (explicitly via API, or implicitly during a failed read request).

This mechanism can be enabled for NvM blocks via:

- `NvMBlockUseCRCCompMechanism` (AUTOSAR standard)
- `NvMBlockUseMACCompMechanism` (MICROSAR addition)

NvM implementation of the CRC Compare Mechanism feature deviates from AUTOSAR [2]: CRC Compare Mechanism does not depend on Calc Ram Block CRC feature, both features use independent buffers.

Caution
Keep in mind that it is possible to calculate the same CRC value for different data. In that case NvM won't write and the NV memory content is not up to date with RAM content – user must be able to operate with outdated data (after e.g. shutdown and startup).

Caution
Please note that you still need to use the `NvM_SetRamBlockStatus` function! During a `WriteAll` NvM will only check the CRC, if the block status is valid and changed – `NvM_SetRamBlockStatus` was invoked with "true".

Caution
This feature affects not only the `NvM_WriteAll`, where writing depends on RAM block status, but also the `NvM_WriteBlock/NvM_WritePRAMBlock` job.

If enabled, the NV write may be skipped.

If disabled, the above APIs will always write the data.

Note
Same data for multiple datasets won't be written, if this feature is enabled, because after the first dataset write, the DataIntegrityRecord matches and NvM will skip the write to NV RAM. If writing is required, enable the chapter 2.1.1.1 Block Id check feature (currently only supported for CRC).

2.4.21.1 Redundant blocks

For redundant blocks the handling was extended to ensure data redundancy in NV RAM: NvM will first check the readability of both subsequent blocks (i.e. reading one byte) and then, depending on the read result:

- Both blocks readable: behave as for other block types and check if writing to NV can be skipped
- One of the blocks (or both) not readable: do not check if writing can be skipped and ensure that block will be written to restore redundancy.

Note
NvM will never compare any data or check its content when doing this readability check and a successful read will be interpreted as valid data, even though it may be outdated.

2.4.22 Check loss of redundancy

The configuration option `NvMCheckLossOfRedundancy` can be enabled on a per block basis for all redundant blocks which have a data integrity mechanism configured. If this feature is used for a block, both data instances of the block will be checked for consistency on a read request.

Should their data integrity record not be equal (i.e. the data is no longer redundant) or should one of the sub blocks be defect, a call to a function specified in `NvM/NvMCommon/NvMDetectedLossOfRedundancyCallback` will be made. The callback includes the ID of the affected block, so actions can be derived on a per block basis. Should at least one of the instances be valid, the result of the job will still be successful, the callback will only be called in addition to that.

Please keep in mind that a detected loss of redundancy implies that there is valid data available and the job result of this block will be NVM_REQ_OK. If you want to restore the blocks redundancy you will have to make sure the block will be written (either `NvM_WriteBlock` or `NvM_SetRamBlockStatus`). In case that no data could be read this feature has no effect.

Note
Please note that this feature can be considered an alternative to the API `NvM_RepairRedundantBlock`. Additionally keep in mind, that there are no synergy effects between these features.

Note
This feature only works when a data integrity mechanism is enabled for the configured block, since the mechanism of detecting lost redundant data works by verifying the data integrity record of the data.

Caution
Enabling this feature will negatively impact the read performance depending on the amount of errors in the image.

Caution
The callback happens before the queued job is completed. This means it is not possible to directly queue a write job to fix the loss of redundancy, because the NvM would reject it due to the current `NVM_REQ_PENDING` job status.

2.4.23 Ciphering interface

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

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

- Write data: NvM encrypts the plain user data and then forwards the ciphered data to the device.
- Read data: NvM reads the ciphered data from device, decrypts the data and finally provides the plain data to the user.

To check the integrity of the ciphered data a data integrity mechanism can be configured. NvM will then calculate the data integrity record over encrypted data and recalculate and validate the integrity record before decryption: the data integrity record always matches the ciphered data.

If the integrity check fails, NvM will not decrypt the data but abort the job with `NVM_REQ_INTEGRITY_FAILED`.

Caution
Usage of NvMCalcRamBlockCrc and data ciphering is not recommended: when invoking the `NvM_SetRamBlockStatus` the NvM will calculate the CRC in background – at this point the NvM only knows the user data and therefore calculates the CRC over the user data. The actual CRC will be calculated over the ciphered data later during a write request – two different data, one CRC based functionality – NvM might read the ciphered data from NV RAM though the user data did not change.

2.4.23.1 CSM usage

Since the NvM currently supports only synchronous encryption and decryption, NvM relies directly on `Csm_Encrypt` and `Csm_Decrypt` outcome.

To avoid frequent read and write job abortions the NvM supports a retry mechanism.

Overview of CSM job outcome and associated NvM handling:

CSM job outcome	NvM behavior
Return code: - CSM_E_OK	NvM continues job processing. This behavior is the same for read and write (decryption and encryption).
Return code: - CSM_E_BUSY - CSM_E_QUEUE_FULL	NvM attempts to retry the CSM job in the next main function cycle. Exceeded retry attempts will lead to job abortion and the NvM result `NVM_REQ_NOT_OK`. For details see 2.4.23.2.2.
All other return codes	NvM classifies this as an error. This will lead to job abortion and the NvM result `NVM_REQ_NOT_OK`.
CSM job result length does not fit the configured values (see 2.4.23.2).	

Table 2-6 NvM – CSM job result mapping

To store the ciphered data NvM uses separate internal “CSM buffer” (another than used for e.g. Explicit Synchronization Mechanism 2.4.20):

> Encryption: input buffer is the user data buffer (provided by user, or the NvM internal buffer), output buffer is the “CSM buffer”,

> Decryption: input buffer the is “CSM buffer”, output buffer is the user data buffer (provided by the user or the NvM internal buffer).

Caution
Depending on configuration and NvM job setup, NvM may pass the user buffer directly to the CSM module to store the decrypted data in. If the CSM job fails, NvM does not guarantee the buffer content, CSM or the cryptographic stack may modify the content before failing!

2.4.23.2 Configuration

2.4.23.2.1 Block configuration

For each NvM block the following parameters must be configured:

- NvMCsmEncryptionJobReference: reference to a CSM encryption job
- NvMCsmDecryptionJobReference: reference to a CSM decryption job
- NvMNvBlockNVRAMDataLength: size of the ciphered data after encryption and before decryption (data size stored in NV RAM).

Note

- NvMNvBlockLength marks the user data length,
- NvMNvBlockNVRAMDataLength marks the NV RAM data length, in this case this is the ciphered data length. The result length of the ciphered data must match this value (see 2.4.23.1). Because of 16 byte aligned algorithm support NvM assumes this length to be the aligned NvMNvBlockLength or even higher.

  The highest NvMNvBlockNVRAMDataLength + the length of the data integrity record will be used as the size of the internal “CSM buffer”.

2.4.23.2.2 General configuration

In the NvMCommon configuration container the parameter NvMCsmRetryCounter can be configured. This parameter will be used to limit the retry attempts for CSM jobs that are rejected because of a busy state of the CSM.

Configuring this parameter to value 0 means: no retry behavior in error case – the job will be aborted directly.

2.5 Error Handling

2.5.1 Development Error Reporting

By default, development errors are reported to the DET using the service `Det_ReportError()` as specified in [3], if development error reporting is enabled (i.e. pre-compile parameter `NVM_DEV_ERROR_DETECT == STD_ON`).

The reported NVM ID can be seen here [chapter 1].

The reported service IDs identify the services which are described in 4.4. The following table presents the service IDs and the related services:

Service ID	Service
0x00	`NvM_Init()`
0x01	`NvM_SetDataIndex()`
0x02	`NvM_GetDataIndex()`
0x03	`NvM_SetBlockProtection()`
0x04	`NvM_GetErrorStatus()`
0x05	`NvM_SetRamBlockStatus()`

Service ID	Service
0x06	NvM_ReadBlock()/NvM_ReadPRAMBlock()
0x07	NvM_WriteBlock()/NvM_WritePRAMBlock()
0x08	NvM_RestoreBlockDefaults()/NvM_RestorePRAMBlockDefaults()
0x09	NvM_EraseNvBlock()
0x0A	NvM_CancelWriteAll()
0x0B	NvM_InvalidateNvBlock()
0x0C	NvM_ReadAll()
0x0D	NvM_WriteAll()
0x0E	NvM_MainFunction()
0x0F	NvM_GetVersionInfo()
0x10	NvM_CancelJobs()
0x13	NvM_SetBlockLockStatus()
0x14	NvM_KillWriteAll()
0x15	NvM_RepairRedundantBlocks()
0x16	NvM_KillReadAll()
0x17	NvM_PreReadAll()

Table 2-7 Mapping of service IDs to services

Note

The Service ID which is passed to the DET when the PRAM-API-function is called, is the ID of the corresponding non-PRAM-function.

The errors reported to DET are described in the following table:

Error Code		Description
0x14	NVM_E_NOT_INITIALIZED	Every API service, except NvM_Init() and NvM_GetVersionInfo(), may check if NVM has already been initialized.
0x15	NVM_E_BLOCK_PENDING	As long as an asynchronous operation on a certain Block has not been completed, no further requests belonging to this Block are allowed.
0x18	NVM_E_BLOCK_CONFIG	This service is not possible with this configuration.
0x0A	NVM_E_PARAM_BLOCK_ID	NVM API services may check, whether the passed BlockId is in the allowed range.
0x0B	NVM_E_PARAM_BLOCK_ TYPE	NvM_SetDataIndex() and NvM_GetDataIndex() are restricted to Dataset bocks. If these functions are called with any other bock type, this error code is produced. NvM_RestoreBlockDefaults() is restricted to

Error Code		Description
		blocks configured with ROM defaults or an init callback.
0x0C	NVM_E_PARAM_BLOCK_DATA_IDX	NvM_SetDataIndex() may check the range of the passed DataIndex.
0x0D	NVM_E_PARAM_ADDRESS	A wrong pointer parameter was passed. (NULL_PTR passed in an asynchronous call, e.g. NvM_WriteBlock() for a non-permanent block)
0x0E	NVM_E_PARAM_DATA	A NULL_PTR was passed in one of the synchronous functions NvM_GetDataIndex(), NvM_GetErrorStatus() or NvM_GetVersionInfo().

Table 2-8 Errors reported to DET

2.5.1.1 Parameter Checking

AUTOSAR requires that API functions check the validity of their parameters. The checks in Table 2-9 are internal parameter checks of the API functions.

The following table shows which parameter checks are performed on which services:

Service \ Check	Module's initialization Status check	Block's Management Type check	Block's Pending State check	Block Id check	DataIndex check	Pointers check
NvM_Init()						
NvM_SetDataIndex()	■	■	■	■	■	
NvM_GetDataIndex()	■	■		■		■
NvM_SetBlockProtection()	■		■	■		
NvM_GetErrorStatus()	■			■		■
NvM_GetVersionInfo()						■
NvM_SetRamBlockStatus()	■	■	■	■	■	
NvM_SetBlockLockStatus()	■			■		
NvM_ReadBlock()	■			■	■	■
NvM_ReadPRAMBlock()	■			■	■	■
NvM_WriteBlock()	■			■	■	■
NvM_WritePRAMBlock()	■			■	■	■
NvM_RestoreBlockDefaults()	■	■		■		■
NvM_RestorePRAMBlockDefaults()	■	■		■		■
NvM_EraseNvBlock()	■			■	■	
NvM_CancelWriteAll()	■					
NvM_InvalidateNvBlock()	■			■	■	

Service \ Check	Module's initialization Status check	Block's Management Type check	Block's Pending State check	Block Id check	DataIndex check	Pointers check
NvM_ReadAll()	■		■			
NvM_WriteAll()	■		■			
NvM_MainFunction()	■					
NvM_CancelJobs()	■			■		
NvM_RepairRedundantBlocks()	■					
NvM_KillWriteAll()	■					
NvM_KillReadAll()	■					
NvM_PreReadAll()	■		■			

Table 2-9 Development Error Checking: Assignment of checks to services

2.5.2 Production Code Error Reporting

NvM checks and reports following error codes to DEM:

Error Code	Description
NVM_E_INTEGRITY_FAILED	API request integrity failed
NVM_E_REQ_FAILED	API request failed
NVM_E_WRITE_PROTECTED	NvM_WriteBlock, NvM_EraseNvBlock and NvM_InvalidateNvBlock check, if the block with specified BlockId is write-protected, before it is written (or erased or invalidated).
NVM_E_QUEUE_OVERFLOW	All asynchronous requests can only be enqueued if the queue is not full.
NVM_E_LOSS_OF_REDUNDANCY	One single block of a redundant block is invalid.

Table 2-10 Errors reported to DEM

According to AUTOSAR component specific DEM error codes must be configured in DEM, NvM configuration references them.

To report production errors to DEM, NvM uses the Dem_ReportErrorStatus API which has to be published via Dem.h (included if NvM references at least one DEM error code). NvM invokes the DEM API with configured error code and the status

`DEM_EVENT_STATUS_FAILED`. NvM never uses the status `DEM_EVENT_STATUS_PASSED`.

Caution
NvM does not report DEM errors without reference to DEM. If any DEM error reporting is required, the NvM error code has to point to a DEM error code.

For more information about DEM see [4]

2.5.3 Compile-time Block Length Checks

For each block with permanent RAM or ROM, NvM provides the Block Length Check to ensure the configured block length and the permanent RAM's/ROM's length fits to each other.

There are three different checks for permanent RAM

- Automatic Block Length enabled (see 2.1.2): size of permanent RAM must not exceed the configured block length
- Strict Block Length: configured block length has to match the size of permanent RAM exactly
- Non-strict Block Length: configured block length must not exceed the size of permanent RAM

And one for permanent ROM

- Non-strict Block Length: configured block length must not exceed the size of permanent ROM

Basic Knowledge
To check the block length during compile time, NvM uses bitfields – those have to be initialized with positive length. Once a bitfield is initialized with negative length (block length check failed), compiler error shall occur and mark the corresponding line.

Each length check shows all required information: block name, RAM symbol and what is wrong with the block (depends on strict, non-strict or automatic block length)

Note
The Length Check is not required by the AUTOSAR specification of the NVM. It is only provided by the NVM to help to integrate it. The Length Check uses the `sizeof()` operator. Therefore the struct sizes are dependent of the compiler version, the padding configuration and the alignment configuration.

3 Integration

This chapter gives necessary information for the integration of the MICROSAR NVM into an application environment of an ECU.

3.1 Embedded Implementation

The delivery of the NvM contains these source code files:

File Name	Description	Integration Tasks
NvM.c	This is the source file of the NvM.	-
NvM.h	This is the header file of the NvM. Defines the interface of NVM. Only this file shall be included by the application/ user.	-
NvM_Act[.c/.h], NvM_Crc[.c/.h], NvM_JobProc[.c/.h], NvM_Qry[.c/.h] NvM_Queue[.c/.h], NvM_DataIntegrity[.c/.h], NvM_Mac[.c/.h], NvM_MemIfAbstraction[.c/.h], NvM_MemIfMemoryCore[.c/.h], NvM_IntTypes.h	These are files for internal use of the NvM. They must not be accessed by application/ user.	-
NvM_Cbk.h	Contains the declarations of the callback functions being invoked by the underlying device abstraction in case polling mode is disabled.	-
NvM_Types.h	Contains public NvM types and their values (as macros), e.g. the request results. This file is not intended to be included by user/ application. Nevertheless, it is possible to be included directly if required in certain circumstance.	-
NvM_Cfg.c	This is a generated source file containing the NvM configuration, e.g. the configured blocks.	-
NvM_Cfg.h	This is a generated header file containing public configuration parameters. The file will be included by NvM.h and therefore provided to the application/ user.	-
NvM_PrivateCfg.h	This is a generated header file containing private configuration parameters.	-
NvM_MemMap.h	This is a generated header file containing memory mapping sections.	-

Table 3-1 NvM files

3.2 Critical Sections

To protect critical code against interruptions NvM uses following critical section:

> NvM_NVM_EXCLUSIVE_AREA_0

3.3 Compiler Abstraction and Memory Mapping

The objects (e.g. variables, functions, constants) are declared by compiler independent definitions – the compiler abstraction definitions. Each compiler abstraction definition is assigned to a memory section.

The following table contains the memory section names and the compiler abstraction definitions of NVM and illustrates the relationship among each them.

Compiler Abstraction Definitions / Memory Mapping Sections	NVM_PRIVATE_CODE	NVM_PRIVATE_CONST	NVM_PRIVATE_DATA	NVM_FAST_DATA	NVM_PUBLIC_CODE	NVM_PUBLIC_CONST	NVM_APPL_CODE	NVM_APPL_CONST	NVM_APPL_DATA	NVM_CONFIG_CONST	NVM_CONFIG_DATA
NVM_START_SEC_CODE	■		■		■						
NVM_START_SEC_ VAR_NO_INIT_UNSPECIFIED			■								■
NVM_START_SEC_VAR_NO_INIT_8			■								
NVM_START_SEC_VAR_CLEARED_UNSPECIFIED			■								
NVM_START_SEC_VAR_FAST_NO_INIT_8				■							
NVM_START_SEC_ CONST_UNSPECIFIED		■									
NVM_START_SEC_CONST_16		■								■	
NVM_START_SEC_VAR_CLEARED_UNSPECIFIED						■				■	■

Table 3-2 Compiler abstraction and memory mapping

For each start keyword, there is a stop keyword. As these stop keywords are used to restore the default section, the stop keywords do not need to be configured.

Above listed section keywords are compiler dependent. They are set in the files MemMap.h and Compiler.h/Compiler_Cfg.h. Compiler pragmas may be used to open and close a special memory section.

Note
The integrator has to make sure specific section related settings (`#pragma`s) cover **all** (intended) variables defined within in a particular specific section. (It might be desired, and possible with a compiler to further divide variables, e.g. by type/size/alignment requirements).

Expert Knowledge
It is important to understand that a variable declaration lacking an initializer does not actually mean an uninitialized variable (unless the variable has *automatic storage duration)*.

Instead, according to ANSI/ISO, every *object that has static storage duration is not initialized explicitly* (ISO/IEC 9899:1999 chapter 6.7.8 clause 10) shall **be initialized** (according to the rules defined there). Technically, they shall be initialized to 0

However, how a compiler achieves it, is beyond the standard. It is also beyond the standard, how compilers map variables to sections, what default sections they define, etc.

A compiler may treat a variable explicitly initialized to 0 like an “uninitialized” variable, it may treat it like an initialized variable, or it may even treat it completely differently (e.g. some compilers can be setup to emit all explicitly initialized variables to a section “.zbss”, in contrast to “.data” and “.bss”, used for initialized, and uninitialized variables, respectively”.

Therefore, any section definition (`#pragma`s) should consider all variables (regardless of existence of an explicit initializer, and/or eventually other differentiations a compiler might provide), unless there’s a good reason to exclude some of them.

Caution
The sections mentioned above have to fit to the linker configuration (linker command file) as well as to the memory modifier settings in the Compiler Abstraction!

3.4 Dependencies on SW Modules

3.4.1 OSEK / AUTOSAR OS

An OS environment is not necessary unless it is used for interrupt or resource locking issues.

3.4.2 DEM

NvM reports runtime errors to DEM. For more information see chapter 2.5.2.

3.4.3 DET

Module DET: Can be used in development mode. It records all development errors for evaluation purposes. Its usage can be enabled/disabled via configuration tool by the switch **Development Error Reporting**.

3.4.4 MEMIF

The NVM uses configuration parameters defined by the MEMIF.

3.4.5 CRC Library

For CRC calculations the NVM uses the services provided by an AUTOSAR compliant CRC Library.

Note
Since the **Configuration Id Block** must be configured with either CRC16 or CRC32; you will always need the CRC library.

3.4.6 Callback Functions

MICROSAR NVM offers the usage of notifications that can be mapped to callback functions provided by other modules, in order to inform them about job completion. For each NVRAM block a separate callback function may be defined by application. These callback function declarations must be made within the application and be included by the NVM.

3.4.7 RTE

When at least one Service Port is enabled and corresponding PIM (see Technical Reference of RTE) is available, all additional necessary header files are included automatically. SWC must not include `NvM.h`.

3.4.8 BSWM

If the switch **BSWM Multi Block Job Status Information** is enabled the NVM shall inform the BSWM about the current state of a multi block job via `BswM_NvM_CurrentJobMode()`. The multi job callback is not called.

Note
During calling `BswM_NvM_CurrentJobMode()`, if called with status `NVM_REQ_PENDING`, callback related block is still busy. No request for it shall be issued as long as block is busy.

If the switch **BSWM Block Status Information** for a single block is true, the NVM shall inform the BSWM about the current state of the block via `BswM_NvM_CurrentBlockMode()`.

Note
During calling `BswM_NvM_CurrentBlockMode()`, if called with status `NVM_REQ_PENDING`, callback related block is still busy. No request for it shall be issued as long as block is busy.

3.4.9 CSM

The NvM uses the CSM to enable different features.

- **Ciphering**
  For ciphering, the CSM is used to encrypt data before writing and decrypt data after reading it: NV RAM stores ciphered data, user works with plain data. For more information see chapter 2.4.22.
- **MAC**
  To enable MAC as data integrity mechanism, the CSM is used to generated and verify MACs. For more information see chapter 2.4.5.5.2.

3.5 Integration Steps

To integrate MICROSAR NVM into your system, several steps beginning with configuration have to be done:

> Configure MICROSAR NVM and MICROSAR MEMIF according to applications' requirements using MICROSAR configuration tool or a GCE editor.

> Generate the configuration files of the modules NVM and MEMIF.

> Configure and generate the lower modules FEE/EA and the driver modules for FLS/EEP.

> If a FEE or EA module is used that is not delivered by Vector, make sure that the parameters that are exchanged between the two modules are consistent.

> Each application is responsible to make their RAM and ROM blocks available (do not use the static modifier!). The MICROSAR NVM includes the file that declares these blocks and defines memory modifier to address the blocks. This memory modifier can be changed in the `Compiler.h`.

> Make sure all applications using MICROSAR NVM include `Std_Types.h` and `NvM.h` (in that order).

> Check the initialization of the drivers FLS/EEP, FEE/EA and the MICROSAR NVM (MICROSAR NVM does not initialize any other module).

> Make sure that the initialization sequence is correct. FEE/EA and FLS/EEP must be initialized before any NVM request (usually `NvM_ReadAll()`) can be used. Take care initialization sequence of FEE/EA must be finished until FEE/EA is able to accept a job from NvM. In case Fee_MainFunction calls and/or Fls_MainFunction calls are necessary to finish initialization process for FEE/EA the calls have to be executed before NvM requests the first job to FEE/EA.

> Ensure that the main functions of the NVM, the FEE/EA and the FLS/EEP drivers are called cyclically. This must be done within an application task running at sufficient priority (to avoid starving).

> Ensure that a waiting task frees CPU to make it possible that the action for the task is waiting for, can be done!

Finally: Compile and link your MICROSAR NVM together with your project.

3.6 Estimating Resource Consumption

Besides resources needed anyway when using NVM, there are some configuration options influencing resource consumption of your system. In general these options affect usage independently of the number of configured NVRAM blocks. Additionally, each NVRAM block requires resources in RAM, ROM and NV, respectively. The following sections will summarize the options and give you hints, how to estimate their effects.

3.6.1 RAM Usage

In general, each NVRAM block consumes RAM – for the application-defined RAM-block as well as for the internal block management structure, which holds information about request results, blocks' attributes and its current data index. The amount of RAM occupied by the RAM block itself should be equal to the configured length. However, the actual size depends on the size of the object (variable) the application declares. The size of each management area is currently 3 bytes.

However, though they need to be considered when estimating (overall) RAM consumption, RAM blocks technically belong to the clients of NVM.

The configuration options affecting RAM consumption pertain to size of the queue(s) and the option job prioritization. The size of one queue entry depends on the target platform and the compiler options used. It ranges from 8 bytes (16-bit platform, 16-bit pointers) to 12 bytes (32bit architectures, aligned structure members).

Additionally, the setting `NvMDataIntegrityIntBuffer` affects RAM usage: If enabled, the NVM internally allocates a RAM buffer. Its size is at least the size of largest NVRAM block configured with a data integrity mechanism, including the size of the largest data integrity record. Sizes of NVRAM Blocks configured with `NvMBlockUseSyncMechanism`, will also be considered in calculation of internal buffer's size.

Additionally, if a block has a data integrity mechanism configured and the `NvMDataIntegrityIntBuffer` is enabled, an additional buffer with the size of the data integrity record is created for the block.

3.6.2 ROM Usage

Because each NVRAM block's configuration is compiled into a constant block descriptor, the ROM needed is also affected by the whole number of configured NVRAM blocks. Again, the size of one descriptor varies with the target platform and the compiler options used.

There are some configuration options affecting NVM code size. The options

- Development mode
- API configuration class
- use Version Info API
- use Set Ram Block Status API

result in switching on/off complete code sections.

NVM's ROM usage **does not** depend on block configured with ROM defaults. ROM default blocks (defining default data) belong to the clients of NVM, as any callbacks do.

3.6.3 NV Usage

The requirements on NV memory space per device are affected by the NVRAM blocks and their configuration. Basically, each NV block allocates as many bytes as specified for its length, plus data integrity bytes (if configured). Underlying components (FEE or EA) would also add internal management information, as well as padding bytes to meet NV memory device's alignment requirements.

According to the management type of the NVRAM block, it consists of one or more blocks consuming NV space:

- NATIVE 1 NV Block
- REDUNDANT 2 NV Blocks
- DATASET **Count** NV Blocks

3.7 How-To: Integrate NVM with AUTOSAR3 SWC's

Embedded Interface of ASR4 NVM is NOT compatible with ASR3; especially return types have been changed.
However, RTE encapsulates all of them: If an SWC calls a C/S-Interface's operation (via RTE), it always gets `Std_ReturnType`.
Finally, existing embedded code – SWCs as well as NVM itself – compiles against these changed interfaces without modifications.
Unfortunately, to achieve this embedded compatibility, SWC-descriptions (which instruct the RTE generator, how to create compatible code) slightly differ between AUTOSAR services. Users will have to adapt their clients' interface references in order to use AUTOSAR4 BSW along with AUTOSAR3 SWCs.

3.7.1 NVM's provided Interfaces/Ports.

Every interface used by client SWCs needs to be remapped.

3.7.1.1 NvMAdministration

The only operation – `SetBlockProtection` – changed from `POSSIBLE-ERRORS()` to `POSSIBLE-ERRORS(E_NOT_OK)`.
This definition may be exchanged at R-Port side, because the embedded software already used `Std_ReturnType` (and `E_OK`/`E_NOT_OK`), due to RTE API. Code should have been implemented in a defensive way, i.e. it should check return values.
However, this operation can only fail, if development error detection was enabled.

3.7.1.2 NvMService_AC[1|2|3][_SRBS][_Defs]

For information about naming, please refer to 5.1.4
Return types (`POSSIBLE-ERRORS`) of following operations changed:

- `GetErrorStatus`
- `GetDataIndex`
- `SetDataIndex`
- `SetRamBlockStatus`

Note
`NvM_SetDataIndex` and `NvM_GetDataIndex` may fail if “Development Error Detection” is disabled.

Similar to NvMAdministration interface, clients’ R-Port Prototypes must be associated with these new interfaces. The implications on Runnables’ implementations are the same as above – no changes are necessary.

3.7.2 Callbacks (Ports provided by client SWCs)

Actually, callbacks specifications did not change from AUTOSAR3 to AUTOSAR4.

However, a recent feature added to DaVinci Developer and RTE Generator allows for more flexibility in modeling and implementing callback’ signatures. Refer to chapter 5.1.5 for information the relationship between modelled callbacks (SWC’s P-Ports) and their RUNNABLEs’ prototypes.

3.7.3 Request Result Types

In AUTOSAR4 new values for `NvM_RequestResultType` have been defined, namely `NVM_REQ_REDUNDANCY_FAILED`.
However, since the actual usage is not specified, it will not be used by NVM; its interface description omits it, and clients do not need to deal with it.

The NvM uses `NVM_REQ_RESTORED_FROM_ROM` for blocks where the default ROM data was loaded during `NvM_ReadBlock()`, `NvM_ReadAll()` or `NvM_PreReadAll()`.
This job result can be mapped in two different ways for AUTOSAR3:

- Map it to `NVM_REQ_OK` when the data shall be used normally. In this case the application does not need to know whether default data is configured.
- Map it to the fail handling of, i.e. all other results than `NVM_REQ_OK`. The NvM will always load the default data in this case (when configured). So the application can implement special handling for this case.

4 API Description

4.1 Interfaces Overview

For an interfaces overview please see Figure 1-3.

4.2 Definitions

Type Name	C-Type	Description	Value Range
NvM_RequestResultType	uint8	An asynchronous API service can have following results or status that can be polled by NvM_GetErrorStatus().	NVM_REQ_OK The last asynchronous request has been finished successfully. This is the default value after reset. This status has the value 0. Can be delivered by all asynchronous APIs.
			NVM_REQ_NOT_OK The last asynchronous request has been finished unsuccessfully. Can be delivered by all asynchronous APIs.
			NVM_REQ_PENDING An asynchronous request has already been queued. Can be delivered by all asynchronous APIs.
			NVM_REQ_INTEGRITY_FAILED A NV block was supposed to be valid but it turned out that the data are corrupted (either data integrity validation fails or the FEE or the EA reported an inconsistency). Can be delivered by NvM_ReadBlock, NvM_ReadAll or NvM_PreReadAll.
			NVM_REQ_BLOCK_SKIPPED The block was skipped during a multi block request. Can be delivered by NvM_ReadAll, NvM_WriteAll and NvM_PreReadAll.
			NVM_REQ_NV_INVALIDATED The NV block is marked as invalid. Can be delivered by NvM_ReadBlock, NvM_ReadAll and NvM_PreReadAll.
			NVM_REQ_CANCELLED The last asynchronous NvM_WriteAll() has been cancelled by NvM_CancelWriteAll().
			NVM_REQ_RESTORED_FROM_ROM Read of the NV block did result in loading the default data to the RAM image. Can be delivered by

Type Name	C-Type	Description	Value Range
			NvM_ReadBlock(), NvM_ReadAll() or NvM_PreReadAll().
NvM_BlockIdType	uint16	It is the type of a block handle that is used by the application in order to access a NVM block. There are two reserved IDs: > Block ID 0 for multi block requests (Block ID 0 is only allowed for API NvM_GetErrorStatus()) and > Block ID 1 for the configuration Id block The block handles are created as defines in an ascending define list which can be found in NvM_Cfg.h. Define pattern: NvMConf_NvMBlockDescriptor_<BlockName>	$\left\{ \begin{matrix} [0..2^n[,]2^{15}..2^{15}+2^n[, \\ n = 16 - \text{NVM\_DATASET\_SELECTION\_BITS} \end{matrix} \right\}$ NVM_DATASET_SELECTION_BITS is the maximum number of bits that are needed in order to store the maximum dataset value. The second range describes each block's DCM alias. Block ID 0 does not have such an alias. Example: The dataset block with the greatest number of datasets has six of them. So it is necessary to store the data index 0…5 to select the appropriate dataset block. To store the value five, three bits are necessary. So NVM_DATASET_SELECTION_BITS has the value 3. This means that only the block IDs 0 … 8191 are available as block handles. Additionally NVM provides access to these IDs' block aliases via handles 32768+1 … 32768+8191
NvM_ServiceIdType	uint8	Service Ids of the different service routines of the NVM.	NVM_INIT (0u) NVM_SET_DATA_INDEX (1u) NVM_GET_DATA_INDEX (2u) NVM_SET_BLOCK_PROTECTION (3u) NVM_GET_ERROR_STATUS (4u) NVM_SET_RAM_BLOCK_STATUS (5u) NVM_READ_BLOCK (6u) NVM_WRITE_BLOCK (7u) NVM_RESTORE_BLOCK_DEFAULTS (8u) NVM_ERASE_BLOCK (9u) NVM_CANCEL_WRITE_ALL (10u) NVM_INVALIDATE_NV_BLOCK (11u) NVM_READ_ALL (12u) NVM_WRITE_ALL (13u) NVM_MAINFUNCTION (14u) NVM_GET_VERSION_INFO (15u) NvM_CANCEL_JOBS (16u)

Type Name	C-Type	Description	Value Range
			NVM_SET_BLOCK_LOCK_STATUS (19u) NVM_KILL_WRITE_ALL (20u) NVM_REPAIR_REDUNDANT_BLOCKS (21u) NVM_KILL_READ_ALL (22u) NVM_PRE_READ_ALL (23u) The single values are applied as defines. See also chapter 2.5.1

Table 4-1 Type definitions

4.3 Global API Constants

These NVM specific constants are available through the inclusion of `NvM.h`. They are configurable within DaVinci Configurator Pro.

- `NVM_COMPILED_CONFIG_ID`: configured identifier for the NV memory layout
- `NVM_NO_OF_BLOCK_IDS`: number of all defined NVRAM Blocks (including reserved blocks)
- Name of the NVRAM blocks

4.4 Services provided by NVM

The NVM API consists of services, which are realized by function calls.

4.4.1 NvM_Init

Prototype	
`void NvM_Init ( void )`	
Parameter	
--	--
Return code	
`void`	--
Functional Description	
Service for basic NVM initialization. The time consuming NVRAM block initialization and setup according to the block descriptor is done by the `NvM_ReadAll` request.	
Particularities and Limitations	
- This service is synchronous. - This service is non re-entrant. - This service is always available.	
Expected Caller Context	

> This service is expected to be called in application context. > It is expected to be exclusively called by ECU State Manager (or a comparable component)

Table 4-2 NvM_Init

4.4.2 NvM_SetDataIndex

Prototype	
`Std_ReturnType NvM_SetDataIndex ( NvM_BlockIdType BlockId, uint8 DataIndex )`	
Parameter	
`BlockId`	The Block identifier.
`DataIndex`	Index position of a Block in the NV Block of Dataset type.
Return code	
`E_OK`	Request has been accepted.
`E_NOT_OK`	Request has not been accepted, e.g. Det error occurred.
Functional Description	
The request sets the specified index to associate a dataset NV block (with/without ROM blocks) with its corresponding RAM block. The `DataIndex` needs to have a valid value before a read/write/erase or invalidate request is initiated. If the dataset block has a set of ROM defaults, this function is used (prior to `NvM_ReadBlock()`) to select the appropriate ROM set.	
Particularities and Limitations	
> This service is synchronous. > This service is re-entrant. > This service is available if API configuration class 2 or 3 is configured. > The NVRAM manager shall have been initialized before this request is called. **Note** Usage of Explicit Synchronization, does not permit NvM's clients to issue new requests for a pending block.	
Expected Caller Context	
> This service is expected to be called in application context.	

Table 4-3 NvM_SetDataIndex

4.4.3 NvM_GetDataIndex

Prototype	
`Std_ReturnType NvM_GetDataIndex ( NvM_BlockIdType BlockId, uint8* DataIndexPtr )`	
Parameter	
`BlockId`	The Block identifier.
`DataIndexPtr`	Address where the current DataIndex shall be written to
Return code	
`E_OK`	Request has been accepted.
`E_NOT_OK`	Request has not been accepted, e.g. Det error occurred.

Functional Description
The request passes the current DataIndex (association) of the specified dataset block.
Particularities and Limitations
> This service is synchronous. > This service is re-entrant. > This service is available if API configuration class 2 or 3 is configured. > The NVRAM manager shall have been initialized before this request is called.
Expected Caller Context
> This service is expected to be called in application context.

Table 4-4 NvM_GetDataIndex

4.4.4 NvM_SetBlockProtection

Prototype	
`Std_ReturnType NvM_SetBlockProtection( NvM_BlockIdType BlockId, boolean ProtectionEnabled )`	
Parameter	
`BlockId`	The Block identifier.
`ProtectionEnabled`	This parameter is responsible for setting the write protection of a selected NVRAM block: `TRUE`: enable protection `FALSE`: disable protection
Return code	
`E_OK`	Request has been accepted.
`E_NOT_OK`	Request has not been accepted, e.g. Det error occurred.
Functional Description	
The request sets the write protection for the NV block. Any further write/erase/invalidate requests to the NVRAM block are rejected synchronously if the NV block-write protection is set. The data area of the RAM block remains writable in any case.	
Particularities and Limitations	
> This service is synchronous. > This service is re-entrant. > This service is available if API configuration class 3 is configured. > The NVRAM Manager shall have been initialized before this request is called. The protection cannot be released for a write once block that has already been written.	
Note Usage of Explicit Synchronization, does not permit NvM's clients to issue new requests for a pending block.	
Expected Caller Context	

> This service is expected to be called in application context.

Table 4-5 NvM_SetBlockProtection

4.4.5 NvM_GetErrorStatus

Prototype	
`Std_ReturnType NvM_GetErrorStatus ( NvM_BlockIdType BlockId, NvM_RequestResultType* RequestResultPtr )`	
Parameter	
`BlockId`	The Block identifier.
`RequestResultPtr`	Pointer where the result shall be written to. Result is of type `NvM_RequestResultType`. All possible results are described in chapter 4.2.
Return code	
`E_OK`	Request has been accepted.
`E_NOT_OK`	Request has not been accepted, e.g. Det error occurred.
Functional Description	
The request reads the block dependent error/status information and writes it to the given address. The status/error information was set by a former or current asynchronous request. This API can also be requested with BlockId 0 (multi block). Then the multi block error/status information will be read to the given address. Only `NvM_ReadAll()`, `NvM_WriteAll()` and `NvM_PreReadAll()` are multi block requests and change the status/error information of the multi block.	
Particularities and Limitations	
> This service is synchronous. > This service is re-entrant. > This service is always available. > The NVRAM Manager shall have been initialized before this request is called.	
Expected Caller Context	
> This service is expected to be called in application context.	

Table 4-6 NvM_GetErrorStatus

4.4.6 NvM_GetVersionInfo

Prototype	
`void NvM_GetVersionInfo ( Std_VersionInfoType* versioninfo )`	
Parameter	
`versioninfo`	Pointer to the address where the version info shall be written to.
Return code	
`void`	--

Functional Description
The request writes the version info (Vendor ID, module ID, SW major version, SW minor version, SW patch version) to the given pointer.
Particularities and Limitations
> This service is synchronous. > This service is re-entrant. > This service is available if the pre-compile switch **Use version info API** is enabled.
Expected Caller Context
> This service is expected to be called in application context.

Table 4-7 NvM_GetVersionInfo

4.4.7 NvM_SetRamBlockStatus

Prototype	
`Std_ReturnType NvM_SetRamBlockStatus ( NvM_BlockIdType BlockId, boolean BlockChanged )`	
Parameter	
`BlockId`	The block identifier.
`BlockChanged`	Sets the new status of the RAM block: TRUE: Validates the RAM block and marks it as changed. If the block has a CRC and the option NVM_CALC_RAM_BLOCK_CRC is TRUE the CRC calculation is initiated. FALSE: Mark the block as unchanged
Return code	
`E_OK`	Request has been accepted.
`E_NOT_OK`	Request has not been accepted, e.g. Det error occurred.
Functional Description	
The request sets a block's status to valid/changed respectively to unchanged. Setting a block to changed marks it for writing it during `NvM_WriteAll()`. If the block shall be set to **changed**, it has a CRC and the option `NVM_CALC_RAM_BLOCK_CRC` is `TRUE` the CRC calculation of the RAM block is initiated.	

Note
Though this service is defined to operate synchronously, the CRC re-calculation will be performed asynchronously. However, there is no restriction on accessing RAM block data, or on calling other services. Consistency of data and CRC is ensured by WriteBlock/WriteAll, which will unconditionally recalculate the CRC before writing. Requesting CRC re-calculation, using `NvM_SetRamBlockStatus` again, will be recognized in a save way, the calculation will be re-queued, if necessary.

Particularities and Limitations
> This service is synchronous. > This service is re-entrant. > This service is always available. > The NVRAM Manager shall have been initialized before this request is called.
Expected Caller Context
> This service is expected to be called in application context.

Table 4-8 NvM_SetRamBlockStatus

4.4.8 NvM_SetBlockLockStatus

Prototype	
`void NvM_SetBlockLockStatus( NvM_BlockIdType BlockId, boolean BlockLocked )`	
Parameter	
`BlockId`	The Block identifier.
`BlockLocked`	This parameter is responsible for setting the lock protection status of a selected NVRAM block: `TRUE`: Lock shall be enabled `FALSE`: Lock shall be disabled
Return code	
–	

Functional Description

Service for setting/resetting the lock of a NV block.

If locked, the NV contents associated to the NVRAM block identified by BlockId, will not be modified by any subsequent write request, i.e. the Block will be skipped during `NvM_WriteAll`; other requests, namely `NvM_WriteBlock`, `NvM_InvalidateNvBlock`, `NvM_EraseNvBlock`, will be rejected without error notification to Det or Dem; i.e. they just return `E_NOT_OK`.

During processing of `NvM_ReadAll`, a locked NVRAM block shall be loaded from NV memory, regardless of RAM block's state. After that the lock is disabled again.

If a block gets locked with `NvM_SetBlockLockStatus`, only the original NVRAM block is locked, regardless which BlockId was passed - original or DCM (see chapter 2.4.19)

Expert Knowledge
It is allowed to use this service for an already pending block. However, setting a lock affects only subsequent requests; an already pending write will be processed.

This is a deviation from AUTOSAR, which prohibits this request for a pending block.

Particularities and Limitations
> This service is synchronous. > This service is re-entrant. > This service is always available independent on API configuration class. > The NVRAM Manager shall have been initialized before this request is called. The protection cannot be released for a write once block that has already been written. > The service is only usable by BSW components; it is not accessible via RTE.
Expected Caller Context
> This service is expected to be called by DCM.

Table 4-9 NvM_SetBlockLockStatus

4.4.9 NvM_MainFunction

Prototype	
`void NvM_MainFunction ( void )`	
Parameter	
--	--
Return code	
`void`	--
Functional Description	
This function has to be called cyclically. It is the entry point of the NVRAM Manager. In here the processing of the asynchronous jobs (read/write/erase/invalidate/CRC calculation…) is handled.	
Particularities and Limitations	
> This service is synchronous. > This service is non re-entrant. > This service is always available. > The NVRAM Manager shall have been initialized before this request is called.	
Expected Caller Context	
> This service is expected to be called in application context.	

Table 4-10 NvM_MainFunction

4.4.10 NvM_ReadBlock

Prototype	
`Std_ReturnType NvM_ReadBlock ( NvM_BlockIdType BlockId,` `void* NvM_DstPtr )`	
Parameter	
`BlockId`	The Block identifier.

NvM_DstPtr	Pointer where the data of a non-permanent RAM block shall be written to. If the block is permanent NULL_PTR shall be passed.
Return code	
E_OK	Request has been accepted.
E_NOT_OK	Request has not been accepted, e.g. because of a list overflow.
Functional Description	
Request to copy the data of the NV block to its corresponding RAM block. This function queues the read request and returns the acceptance result synchronously. The NVM can notify the application by callback when the service is finished.	
Particularities and Limitations	
> This service is asynchronous. > This service is re-entrant. > This service is available if API configuration class 2 or 3 is configured. > The NVRAM Manager shall have been initialized before this request is called. In development mode the service will not accept the call if the block is already queued (either for this or for a different service). **Note** Usage of Explicit Synchronization, does not permit NvM's clients to issue new requests for a pending block.	
Expected Caller Context	
> This service is expected to be called in application context.	

Table 4-11 NvM_ReadBlock

4.4.11 NvM_ReadPRAMBlock

Prototype	
Std_ReturnType NvM_ReadPRAMBlock (NvM_BlockIdType BlockId)	
Parameter	
BlockId	The Block identifier.
Return code	
E_OK	Request has been accepted.
E_NOT_OK	Request has not been accepted, e.g. because of a list overflow.
Functional Description	
Request to copy the data of the NV block to its permanent RAM block. This function is a wrapper API which calls NvM_ReadBlock. For more information refert to the corresponding non-PRAM function.	

Particularities and Limitations
> This service is asynchronous. > This service is re-entrant. > This service is available if API configuration class 2 or 3 is configured. > The NVRAM Manager shall have been initialized before this request is called. In development mode the service will not accept the call if the block is already queued (either for this or for a different service). **Note** Usage of Explicit Synchronization, does not permit NvM's clients to issue new requests for a pending block.
Expected Caller Context
> This service is expected to be called in application context.

Table 4-12 NvM_ReadPRAMBlock

4.4.12 NvM_WriteBlock

Prototype	
`Std_ReturnType NvM_WriteBlock ( NvM_BlockIdType BlockId, const void* NvM_SrcPtr )`	
Parameter	
`BlockId`	The Block identifier.
`NvM_SrcPtr`	Pointer where the data of a non-permanent RAM block shall be read from. If the block is permanent, `NULL_PTR` shall be passed.
Return code	
`E_OK`	Request has been accepted.
`E_NOT_OK`	Request has not been accepted, e.g. list overflow.
Functional Description	
Request for copying data from the RAM block to its corresponding NV block. This function queues the write request and returns the acceptance result synchronously. If the block has CRC, the RAM block CRC will be recalculated before the data and the CRC are written to the NV memory. This is also the case if CRC background calculation (`NvM_SetRamBlockStatus` and `NvMCalcRamBlockCrc`) is configured for the block. If writing the data to NV memory fails, the NVM will retry writing. The number of write retries is a configuration option. The NVM can notify the application by callback when the service is finished.	

Particularities and Limitations
> This service is asynchronous. > This service is re-entrant. > This service is available if API configuration class 2 or 3 is configured. > The NVRAM Manager shall have been initialized before this request is called. In development mode the service will not accept the call if the block is already queued (either for this or for a different service). If the block's write protection is activated, it can't be written. **Note** Usage of Explicit Synchronization, does not permit NvM's clients to issue new requests for a pending block.
Expected Caller Context
> This service is expected to be called in application context.

Table 4-13 NvM_WriteBlock

4.4.13 NvM_WritePRAMBlock

Prototype	
`Std_ReturnType NvM_WritePRAMBlock ( NvM_BlockIdType BlockId)`	
Parameter	
`BlockId`	The Block identifier.
Return code	
`E_OK`	Request has been accepted.
`E_NOT_OK`	Request has not been accepted, e.g. list overflow.
Functional Description	
Request for copying data from the permanent RAM block to its corresponding NV block. This function is a wrapper API which calls `NvM_WriteBlock`. For more information refert to the corresponding non-PRAM function.	
Particularities and Limitations	
> This service is asynchronous. > This service is re-entrant. > This service is available if API configuration class 2 or 3 is configured. > The NVRAM Manager shall have been initialized before this request is called. In development mode the service will not accept the call if the block is already queued (either for this or for a different service). If the block's write protection is activated, it can't be written. **Note** Usage of Explicit Synchronization, does not permit NvM's clients to issue new requests for a pending block.	
Expected Caller Context	

> This service is expected to be called in application context.

Table 4-14 NvM_WriteBlock

4.4.14 NvM_RestoreBlockDefaults

Prototype	
`Std_ReturnType NvM_RestoreBlockDefaults ( NvM_BlockIdType BlockId, void* NvM_DstPtr )`	
Parameter	
`BlockId`	The Block identifier.
`NvM_DstPtr`	Pointer where the data of a non-permanent RAM block shall be written to. If the block is permanent, `NULL_PTR` shall be passed.
Return code	
`E_OK`	Request has been accepted.
`E_NOT_OK`	Request has not been accepted, e.g. list overflow.
Functional Description	
Request to copy the ROM block default data to its corresponding RAM block. The selected block needs either ROM defaults or an initialization callback. This function queues the restore request and returns the acceptance result synchronously. The NVM can notify the application by callback when the service is finished.	
Particularities and Limitations	
> This service is asynchronous. > This service is re-entrant. > This service is available if API configuration class 2 or 3 is configured. > The NVRAM Manager shall have been initialized before this request is called. In development mode the service will not accept the call if the block is already queued (either for this or for a different service). This function is not intended for reading ROM sets of a dataset ROM block. Use `NvM_ReadBlock` instead for these blocks.	
Note Usage of Explicit Synchronization, does not permit NvM's clients to issue new requests for a pending block.	
Expected Caller Context	
> This service is expected to be called in application context.	

Table 4-15 NvM_RestoreBlockDefaults

4.4.15 NvM_RestorePRAMBlockDefaults

Prototype
`Std_ReturnType NvM_RestoreBlockDefaults ( NvM_BlockIdType BlockId)`

Parameter	
BlockId	The Block identifier.
Return code	
E_OK	Request has been accepted.
E_NOT_OK	Request has not been accepted, e.g. list overflow.
Functional Description	
Request to copy the ROM block default data to its permanent RAM block. This function is a wrapper API which calls NvM_RestoreBlockDefaults. For more information refert to the corresponding non-PRAM function.	
Particularities and Limitations	
> This service is asynchronous. > This service is re-entrant. > This service is available if API configuration class 2 or 3 is configured. > The NVRAM Manager shall have been initialized before this request is called. In development mode the service will not accept the call if the block is already queued (either for this or for a different service). This function is not intended for reading ROM sets of a dataset ROM block. Use NvM_ReadBlock instead for these blocks. **Note** Usage of Explicit Synchronization, does not permit NvM's clients to issue new requests for a pending block.	
Expected Caller Context	
> This service is expected to be called in application context.	

Table 4-16 NvM_RestorePRAMBlockDefaults

4.4.16 NvM_EraseNvBlock

Prototype	
Std_ReturnType NvM_EraseNvBlock (NvM_BlockIdType BlockId)	
Parameter	
BlockId	The Block identifier.
Return code	
E_OK	Request has been accepted.
E_NOT_OK	Request has not been accepted, e.g. list overflow.
Functional Description	
Request to erase a specified NV block. This function queues the erase request and returns the acceptance result synchronously. The NVM can notify the application by callback when the service is finished.	

Particularities and Limitations
> This service is asynchronous. > This service is re-entrant. > This service is available if API configuration class 3 is configured. > The NVRAM Manager shall have been initialized before this request is called. In development mode the service will not accept the call if the block is already queued (either for this or for a different service). If the block's write protection is activated it also can't be erased. **Caution** Pay attention that only high priority jobs (priority 0) can be erased! **Note** Usage of Explicit Synchronization, does not permit NvM's clients to issue new requests for a pending block.
Expected Caller Context
> This service is expected to be called in application context.

Table 4-17 NvM_EraseNvBlock

4.4.17 NvM_InvalidateNvBlock

Prototype	
`Std_ReturnType NvM_InvalidateNvBlock ( NvM_BlockIdType BlockId )`	
Parameter	
`BlockId`	The Block identifier.
Return code	
`E_OK`	Request has been accepted.
`E_NOT_OK`	Request has not been accepted, e.g. list overflow.
Functional Description	
Request to invalidate a specified NV block. This function queues the invalidate request and returns the acceptance result synchronously. The NVM can notify the application by callback when the service is finished.	

Particularities and Limitations
> This service is asynchronous. > This service is re-entrant. > This service is available if API configuration class 3 is configured. > The NVRAM Manager shall have been initialized before this request is called. In development mode the service will not accept the call if the block is already queued (either for this or for a different service). If the block's write protection is activated it also can't be invalidated.  **Note** Usage of Explicit Synchronization, does not permit NvM's clients to issue new requests for a pending block.
Expected Caller Context
> This service is expected to be called in application context.

Table 4-18 NvM_InvalidateNvBlock

4.4.18 NvM_ReadAll

Prototype	
`void NvM_ReadAll ( void )`	
Parameter	
--	--
Return code	
`void`	--

Functional Description

Request to (re)load all RAM blocks that have the option `NVM_SELECT_BLOCK_FOR_READALL` selected. The function queues the request that will be processed asynchronously in `NvM_MainFunction`.

Before reloading a block's NV data, it first checks if the RAM block data is still valid. This can only be assured if the block has a checksum. In case of valid RAM data, the NV data will not be reloaded.

Caution
Non-permanent blocks and dataset blocks are also skipped during a ReadAll job.

The first block that is read from NV memory is the configuration ID (block 1). The value is compared to the compiled configuration ID. The result of this check affects the further processing of the ReadAll job, depending on the setting of **Dynamic Configuration Handling**:

In case **Dynamic Configuration Handling** is **disabled**:
The configuration block shall be ignored, therefore it will not be read out at all. The NvM processes the normal runtime preparation for all blocks. That means all blocks are handled normally.

In case **Dynamic Configuration Handling** is **enabled**:
The configuration block is read and checked for mismatch.

> In case the configuration ID matches the current one, the NvM processes the normal runtime preparation for all blocks. That means all blocks are handled normally.

> In case the configuration ID does not match the current one, block could not be read, or the CRC does not match the recalculated one, the NvM behaves following:

- The configuration block is updated and prepared to be rewritten during next multi block write request.
- Blocks which are resistant to changed SW: normal runtime preparation
- Blocks which are not resistant to changed SW: extended runtime preparation. If **Select for ReadAll** was configured for these blocks they will be restored with ROM defaults, if available or the initialization callback is invoked, if configured.

When the last block is reloaded the NVM can notify the application by callback (configurable multi block callback).

Particularities and Limitations

> This service is a multi block request.

> This service is asynchronous.

> This service is non re-entrant.

> This service is always available.

> The NVRAM Manager shall have been initialized before this request is called.

Note
Usage of Explicit Synchronization, does not permit NvM's clients to issue new requests for a pending block.

Expected Caller Context
> This function is intended only to be called by the ECU State Manager during startup.

Table 4-19 NvM_ReadAll

4.4.19 NvM_PreReadAll

Prototype	
`void NvM_PreReadAll ( void )`	
Parameter	
--	--
Return code	
`void`	--
Functional Description	

Request to (re)load all RAM blocks that have the option `NVM_SELECT_BLOCK_FOR_PRE_READALL` selected. The function queues the request that will be processed asynchronously in `NvM_MainFunction`.

Afterwards, NvM is operable for the blocks already read out.This service shall be called before the `NvM_ReadAll()` is executed. The read all is then splitted into two separate phases. The Request Results of all blocks, that are selected for PreReadAll, will be according to the result of the specific block read job. The Request Results of all other blocks, not selected for PreReadAll, will be set to `NVM_REQ_BLOCK_SKIPPED`.

Caution
Non-permanent blocks and dataset blocks are also skipped during a PreReadAll job.

When the last block is reloaded the NVM can notify the application by callback (configurable multi block callback).

Particularities and Limitations
> This service is a multi block request.
> This service is asynchronous.
> This service is non re-entrant.
> The NVRAM Manager shall have been initialized before this request is called.
Expected Caller Context
> This function is intended only to be called by the ECU State Manager during startup.

Table 4-20 NvM_PreReadAll

4.4.20 NvM_WriteAll

Prototype
`void NvM_WriteAll ( void )`

Parameter	
--	--

Return code	
`void`	--

Functional Description

Request to write all blocks with changed RAM data that have the option `NVM_SELECT_BLOCK_FOR_WRITEALL` selected to the NV memory. The function will queue the WriteAll job that will be processed asynchronously.

Note
If the `NvM_SetRamBlockStatus()` API is enabled, blocks are only written during `NvM_WriteAll()` if they were marked as changed previously by calling `NvM_SetRamBlockStatus()` for them.

In case the configuration option `NvMBlockUseSetRamBlockStatus` is disabled for a block, it will always be written during `NvM_WriteAll()` because it can't be detected if the block content changed.

Caution
Non-permanent and dataset blocks will not be written during `NvM_WriteAll()`.

When the last block is written the NVM can notify the application by callback (configurable multiblock callback).

Note
It is not recommended to make any assumption on the order in which blocks will be processed.

It is only ensured that the ConfigID block (ID1) is the final block being processed, in order to "commit" a Configuration Update and any related activity.

Particularities and Limitations

- This service is a multi block request.
- This service is asynchronous.
- This service is non re-entrant.
- This service is always available.
- The NVRAM Manager shall have been initialized before this request is called.

Note
Usage of Explicit Synchronization, does not permit NvM's clients to issue new requests for a pending block.

Expected Caller Context

> This function is intended only to be called by the ECU State Manager during shutdown.

Table 4-21 NvM_WriteAll

4.4.21 NvM_CancelWriteAll

Prototype	
`void NvM_CancelWriteAll ( void )`	
Parameter	
--	--
Return code	
`void`	--
Functional Description	
Request to cancel a running `NvM_WriteAll()` request. This call en-queues the request that will be processed asynchronously.	
Particularities and Limitations	
> This service is asynchronous. > This service is non re-entrant. > This service is always available. > The NVRAM Manager shall have been initialized before this request is called.	
Expected Caller Context	
> This service is expected to be called in application context.	

Table 4-22 NvM_CancelWriteAll

4.4.22 NvM_KillWriteAll

Prototype	
`void NvM_KillWriteAll ( void )`	
Parameter	
--	--
Return code	
`void`	--
Functional Description	
Request to cancel a running `NvM_WriteAll()` request destructively. To keep required wake-up response times in an ECU the ECUM has the possibility to time-out a non-destructive `NvM_CancelWriteAll()` request.	

Particularities and Limitations
> This service is synchronous.
> This service is non re-entrant.
> This service is available if the pre-compile switch **NvmKillWriteAllApi** (only in Generic Editor in container **Nvm_30_CommonVendorParams**) is enabled independent on API configuration class.
> The NVRAM Manager shall have been initialized before this request is called.
Expected Caller Context
> This service is expected to be called by ECUM

Table 4-23 NvM_KilllWriteAll

4.4.23 NvM_CancelJobs

Prototype	
`Std_ReturnType NvM_CancelJobs ( NvM_BlockIdType BlockId)`	
Parameter	
`BlockId`	The Block identifier.
Return code	
`E_OK`	Request has been accepted.
`E_NOT_OK`	Request has not been accepted, e.g. because of a list overflow.
Functional Description	
Request to cancel all jobs pending in the queue for the specified NV Block. Note: Only jobs that are in the waiting queue can be cancelled. An active job therefore cannot be cancelled.	
Particularities and Limitations	
> This service is asynchronous.	
> This service is re-entrant.	
> This service is available if API configuration class 2 or 3 is configured.	
> The NVRAM Manager shall have been initialized before this request is called.	
> Was Cancellation successful Block result is set to `NVM_REQ_CANCELED`.	
Expected Caller Context	
> This service is expected to be called in application context.	

Table 4-24 NvM_CancelJobs

4.4.24 NvM_RepairRedundantBlocks

Prototype	
`void NvM_RepairRedundantBlocks (void)`	
Parameter	
-	-

Return code	
–	-

Functional Description

Request to check the redundancy within NV RAM for all configured redundant blocks. Write protection or lock state does not matter – NvM do not change the data, it always overwrites blocks with data from NV RAM.

If the NvM recognizes a lost redundancy, it will try to restore it via overwriting the defect block with data from valid block.

Nothing to repair:

- Both sub-blocks are readable
- Both sub-blocks' Crcs match the recalculates Crc

Repairable blocks:

- One sub-block isn't readable, another is
- One sub-block's Crc doesn't match the recalculated one, another sub-block's Crc does
- Both sub-blocks' Crcs match the data, but their do not match each other (first block is valid)

Non-repairable blocks:

- Both sub-blocks aren't readable
- Block sub-blocks' Crc does not match the recalculated Crc

NvM will report the error NVM_E_LOSS_OF_REDUNDANCY in case block isn't stored in NV RAM redundantly and the NvM could not restore the redundancy.

Particularities and Limitations

> This service is asynchronous
> This service is re-entrant
> This service is suspendable via all single and multi block requests – it will resume after the requests are done
> This service can be enabled or disabled via configuration
> The NVRAM Manager shall have been initialized before this request is called

Call context

> This service is expected to be called in application context.

Table 4-25 NvM_RepairRedundantBlocks

4.4.25 NvM_KillReadAll

Prototype

```
void NvM_KillReadAll ( void )
```

Parameter	
--	--
Return code	
`void`	--
Functional Description	

Request to abort an ongoing `NvM_ReadAll`. After the request NvM will not access the NV RAM any more (and cancel the underlying module if necessary) to finish the block initialization as soon as possible. NvM still needs `NvM_MainFunction` calls until the multi block error status signals the end of the `NvM_ReadAll`.
 	
NvM behaves as follows:
> Normal `NvM_ReadAll`
> `NvM_KillReadAll`
> For all following blocks:
  > Default data available: load the default data, pretend a successful read
  > Default data not available: block is skipped, although selected for `NvM_ReadAll`
> Multi block error status signals the killed job	

Particularities and Limitations	
> This service is asynchronous. > This service is non re-entrant. > Write once blocks processed after `NvM_KillReadAll` invocation are not write protected. > Status information callbacks and job end notifications are called for all blocks. > The NVRAM Manager shall have been initialized before this request is called.	
Expected Caller Context	
> Any context possible.	

Table 4-26 NvM_KillReadAll

4.5 Services used by NVM

In the following table services provided by other components, which are used by the NVM are listed. For details about prototype and functionality refer to the documentation of the providing component.

Component	API
DET	Det_ReportError
DEM	Dem_SetEventStatus
MEMIF	MemIf_Read
MEMIF	MemIf_InvalidateBlock
MEMIF	MemIf_GetJobResult
MEMIF	MemIf_Write
MEMIF	MemIf_EraseImmediateBlock

Component	API
MEMIF	MemIf_GetStatus
MEMIF	MemIf_Cancel
MEMIF	MemIf_SetMode
CRC	Crc_CalculateCRC16
CRC	Crc_CalculateCRC32
EA	Used by MEMIF
FEE	Used by MEMIF
CSM	Csm_Encrypt Csm_Decypt Csm_MacGenerate Csm_MacVerify

Table 4-27 Services used by the NVM

4.6 Callback Functions

This chapter describes the callback functions that are implemented by the NVM and can be invoked by other modules. The prototypes of the callback functions are provided in the header file `NvM_Cbk.h` by the NVM.

Note
AUTOSAR v4.0.3 and v4.2.2 specified the callbacks to be provided via the NvM_Cbk.h header. Version 4.4 renamed the file to be NvM_MemIf.h but inconsistently defines FEE include source to be NvM.h.

To cover most compatibility MICROSAR NvM provides the callbacks still via NvM_Cbk.h but also implicitly via NvM.h (via including NvM_Cbk.h).

4.6.1 NvM_JobEndNotification

Prototype	
`void NvM_JobEndNotification ( void )`	
Parameter	
-	-
Return code	
`void`	-
Functional Description	
Function to be used by the underlying memory abstraction to signal end of job without error.	
Particularities and Limitations	
> This service is synchronous.	
> This service is non re-entrant.	
Expected Caller Context	

■ The callback function `NvM_JobEndNotification` is intended to be used by the underlying memory abstraction (Fee/Ea) to signal end of job without error.

Table 4-28 NvM_JobEndNotification

4.6.2 NvM_JobErrorNotification

Prototype	
`void NvM_JobErrorNotification ( void )`	
Parameter	
-	-
Return code	
`void`	-
Functional Description	
Function to be used by the underlying memory abstraction to signal end of job with error.	
Particularities and Limitations	
> This service is synchronous. > This service is non re-entrant.	
Expected Caller Context	
> The callback function `NvM_JobErrorNotification` is intended to be used by the underlying memory abstraction (Fee/Ea) to signal end of job with error.	

Table 4-29 NvM_JobErrorNotification

4.7 Configurable Interfaces

At its configurable interfaces the NVM defines notifications that can be mapped to callback functions provided by other modules. The mapping is not statically defined by the BSW module but can be performed at configuration time. The function prototypes that can be used for the configuration have to match the signatures described in the following sub-chapters.

4.7.1 SingleBlockCallbackFunction

Prototype	

`Std_ReturnType <SingleBlockCallbackFunction> (`
`  NvM_ServiceIdType ServiceId,`
`  NvM_RequestResultType JobResult )`	

Parameter	
`ServiceId`	The service identifier (see chapter 4.2) of the completed request. `NvM_ServiceIdType` is of type `uint8`.
`JobResult`	Result of the single block job.
Return code	
`E_OK`	Callback function has been processed successfully
`E_NOT_OK`	Callback function has not been processed successfully.

Functional Description
Callback routine to notify the upper layer that an asynchronous single block request has been finished.
Particularities and Limitations
> This service is synchronous. > This service is non re-entrant. **Caution** This description is limited to embedded code; it does not describe RUNNABLES implementing a callback's behavior in an SWC, but it describes the prototype to be implemented/generated be the RTE or by a BSW component.
Call Context
> Called from `NvM_MainFunction` > Asynchronous block processing completed

Table 4-30 SingleBlockCallbackFunction

4.7.2 SingleBlockCallbackExtendedFunction

Prototype	
`Std_ReturnType <SingleBlockCallbackExtendedFunction> ( NvM_BlockIdType BlockId, NvM_ServiceIdType ServiceId, NvM_RequestResultType JobResult )`	
Parameter	
`BlockId`	The block identifier.
`ServiceId`	The service identifier (see chapter 4.2) of the completed request. `NvM_ServiceIdType` is of type `uint8`.
`JobResult`	Result of the single block job.
Return code	
`E_OK`	Callback function has been processed successfully
`E_NOT_OK`	Callback function has not been processed successfully.

Functional Description
Callback routine to notify the upper layer that an asynchronous single block request has been finished. This callback can be configured additionally to `<SingleBlockCallbackFunction>`. In case that both callbacks are configured, `<SingleBlockCallbackFunction>` will be invoked before `<SingleBlockCallbackExtendedFunction>`. In contradiction to `<SingleBlockCallbackFunction>` the `<SingleBlockCallbackExtendedFunction>` is configured, if the parameter exists and is not `NULL_PTR`. **Note** This callback is an AUTOSAR extension and cannot be used via the option “Use Service Ports”.
Particularities and Limitations
> This service is synchronous. > This service is non re-entrant. **Caution** This description is limited to embedded code; it does not describe RUNNABLES implementing a callback’s behavior in an SWC, but it describes the prototype to be implemented/generated be the RTE or by a BSW component.
Call Context
> Called from `NvM_MainFunction` > Asynchronous block processing completed (during single or multi block request)

Table 4-31 SingleBlockCallbackExtendedFunction

4.7.3 MultiBlockCallbackFunction

Prototype	
`void <MultiBlockCallbackFunction> ( NvM_ServiceIdType ServiceId, NvM_RequestResultType JobResult )`	
Parameter	
`ServiceId`	The service identifier (see chapter 4.2) of the completed request. `NvM_ServiceIdType` is of type `uint8`.
`JobResult`	Result of the multi block job.
Return code	
void	--

Functional Description
Common callback routine to notify the upper layer that an asynchronous multi block request has been finished.
Particularities and Limitations
> This service is synchronous. > This service is non re-entrant.
Call Context
> Called from `NvM_MainFunction`. > Called upon completion of `NvM_ReadAll`, `NvM_WriteAll and NvM_PreReadAll`, respectively

Table 4-32 MultiBlockCallbackFunction

4.7.4 InitBlockCallbackFunction

Prototype	
`Std_ReturnType <InitBlockCallbackFunction> ( void )`	
Parameter	
--	--
Return code	
`E_OK`	Always shall return `E_OK` (ignored by NvM).
Functional Description	
Callback routine which shall be called by the NVM module to copy default data to a RAM block, if no ROM block is configured. **Note** During calling init block callback related block is still busy. No request for it shall be issued as long as block is busy.	
Particularities and Limitations	
> This service is synchronous. > This service is non re-entrant	
Call Context	
> Called from `NvM_MainFunction`	

Table 4-33 InitBlockCallbackFunction

4.7.5 InitBlockCallbackExtendedFunction

Prototype
`Std_ReturnType <InitBlockCallbackExtendedFunction> ( NvM_BlockIdType BlockId, void* BufferPtr, uint16 Length)`

Parameter	
`BlockId`	The block identifier.
`BufferPtr`	Block data pointer to write default data to.
`Length`	The block length.

Return code	
`E_OK`	Always shall return `E_OK` (ignored by NvM).

Functional Description

Callback routine which shall be called by the NVM module to copy default data to a RAM block, if no ROM block is configured.

This callback is the extended version of `<InitBlockCallbackFunction>`.

In contradiction to `<InitBlockCallbackFunction>` the `<InitBlockCallbackExtendedFunction>` is configured, if the parameter exists and is not `NULL_PTR`.

Note
This callback is an AUTOSAR extension and cannot be used via the option “Use Service Ports”.

Note
Only one of the init block callbacks can be configured at a time.

Note
During calling init block callback related block is still busy. No request for it shall be issued as long as block is busy.

Particularities and Limitations

- This service is synchronous.
- This service is non re-entrant

Call Context

- Called from `NvM_MainFunction`
- Called during processing of `NvM_ReadAll or NvM_PreReadAll`, if application shall copy default values into the corresponding RAM block.

Table 4-34 Callback function for Init Block

4.7.6 Callback function for RAM to NvM copy

Prototype	
`Std_ReturnType <NvM_WriteRamBlockToNvm> ( void* NvMBuffer )`	
Parameter	
`NvMBuffer`	Internal RAM mirror where Ram block data shall be written to
Return code	
`E_OK`	Callback function has been processed successfully
`E_NOT_OK`	Callback function has not been processed successfully.
Functional Description	
Block specific callback routine which shall be called in order to let the application copy data from RAM block to internal NvM RAM mirror. **Note** During calling NvM_WriteRamBlockToNvM callback related block is still busy. No request for it shall be issued as long as block is busy.	
Particularities and Limitations	
> This service is synchronous. > This service is non re-entrant	
Call Context	
> Called from `NvM_MainFunction` > Called during processing of NvM write requests, if application shall copy RAM block data into the internal RAM mirror.	

Table 4-35 Callback function for RAM to NvM copy

4.7.7 Callback function for NvM to RAM copy

Prototype	
`Std_ReturnType <NvM_ReadRamBlockFromNvm> ( const void* NvMBuffer )`	
Parameter	
`NvMBuffer`	Internal RAM mirror where Ram block data can be read from
Return code	
`E_OK`	Callback function has been processed successfully
`E_NOT_OK`	Callback function has not been processed successfully.

Functional 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. **Note** During calling NvM_ReadRamBlockFromNvM callback related block is still busy. No request for it shall be issued as long as block is busy.
Particularities and Limitations
> This service is synchronous. > This service is non re-entrant
Call Context
> Called from `NvM_MainFunction` > Called during processing of NvM read requests, if application can copy data from internal RAM mirror to RAM block.

Table 4-36 Callback function for NvM to RAM copy

4.7.8 PreWriteTransformCallbackFunction

Prototype	
`void <PreWriteTransformCallbackFunction> ( NvM_BlockIdType BlockId, void* DataPtr, uint16 Length )`	
Parameter	
`BlockId`	The Block Identifier
`DataPtr`	Pointer to data which shall be written into the NV RAM
`Length`	Configured length of NvM block
Return code	
`void`	--
Functional Description	
Block specific callback routine which shall be called in order to let the application transform the data. For more information see also chapter 2.4.15.	
Particularities and Limitations	
> This service is synchronous. > This service is non re-entrant	
Call Context	
> Called from `NvM_MainFunction` > Called during processing of NvM write requests, if Callback is configured.	

Table 4-37 PreWriteTransformCallbackFunction

4.7.9 PostReadTransformCallbackFunction

Prototype	
`Std_ReturnType <PostReadTransformCallbackFunction> ( NvM_BlockIdType BlockId, void* DataPtr, uint16 Length )`	
Parameter	
`BlockId`	The Block Identifier
`DataPtr`	Pointer to data that has been read from NV RAM
`Length`	Configured length of NvM block
Return code	
`E_OK`	Data is valid
`E_NOT_OK`	Data is not valid, NvM will consider previous read job as unsuccesful
Functional Description	
Block specific callback routine which shall be called in order to let the application transform the data back and verify the data. For more information see also chapter 2.4.15.	
Particularities and Limitations	
> This service is synchronous. > This service is non re-entrant	
Call Context	
> Called from `NvM_MainFunction` > Called during processing of NvM read requests, if Callback is configured.	

Table 4-38 PostReadTransformationCallbackFunction

4.8 Service Ports

Via Service Ports the software components (SWC) have the possibility to execute services of the NVM with an abstract RTE interface. Hence, the software components are independent from the underlying basic software stack.

4.8.1 Client Server Interface

A client server interface is related to a Provide Port (Pport) at the server side and a Require Port (Rport) at client side.

Configuration dependent naming details are described in the chapters 5.1.3 and 5.1.4.

4.8.1.1 Provide Ports on NVM side

At the Pports of the NVM the API functions described in 4.4 are available as Runnable Entities. The Runnable Entities are invoked via Operations. The mapping from a SWC client call to an Operation is performed by the RTE. In this mapping the RTE adds Port Defined Argument Values to the client call of the SWC, if configured.

The following subchapters present the Pports defined for the NVM and their Operations, the API functions related to those Operations and the Port Defined Argument Values to be added by the RTE:

4.8.1.1.1 Padmin_<BlockName>

A port of type **Padmin** is a Pport of one NVRAM block, which is configured to use Service Ports.

If the SWC setting **Long Service Port Names** is enabled, the name of the service ports is Padmin_<BlockName>; if **Long Service Port Names** is disabled, the name is Padmin_<BlockId>.

Available if API Config Class = 3

Operation	API Function	Port Defined Argument Values
SetBlockProtection	`NvM_SetBlockProtection()`	NvM_BlockIdType[4] 1..n

Table 4-39 Operations of Port Prototype Padmin_<BlockName>

4.8.1.1.2 PS_<BlockName>

A port of type **PS** is a Pport of one NVRAM block, which is configured to use Service Ports.

If the SWC setting **Long Service Port Names** is enabled, the name of the service ports is PS_<BlockName>; if **Long Service Port Names** is disabled, the name is PS_<BlockId>.

Operation	API Function	Port Defined Argument Values
GetErrorStatus [1]	`NvM_GetErrorStatus()`	NvM_BlockIdType[4] 1..n
SetRamBlockStatus[1]	`NvM_SetRamBlockStatus()`	NvM_BlockIdType[4] 1..n
SetDataIndex[2,5]	`NvM_SetDataIndex()`	NvM_BlockIdType[4] 1..n
GetDataIndex[2,5]	`NvM_GetDataIndex()`	NvM_BlockIdType[4] 1..n
ReadBlock[2]	`NvM_ReadBlock()`	NvM_BlockIdType[4] 1..n
WriteBlock[2]	`NvM_WriteBlock()`	NvM_BlockIdType[4] 1..n
RestoreBlockDefaults[2, 6]	`NvM_RestoreBlockDefaults()`	NvM_BlockIdType[4] 1..n
EraseBlock[3]	`NvM_EraseNvBlock()`	NvM_BlockIdType[4] 1..n
InvalidateNvBlock[3]	`NvM_InvalidateNvBlock()`	NvM_BlockIdType[4] 1..n

Table 4-40 Operations of Port Prototype PS_<BlockName>

1. Always available
2. Available if API Config Class >= 2
3. Available if API Config Class = 3
4. Is derived from the block's position in the configuration
5. Only available for blocks of Management Type Dataset
6. Only available for blocks with Rom defaults configured

4.8.1.2 Require Ports

NVM invokes callbacks using Rports. These Operations have to be provided by the SWCs by means of Runnable Entities using Pports. These Runnable Entities implement the callback functions expected by the NVM.

The following subchapters present the Require Ports defined for the NVM, the Operations that are called from the NVM and the related Notifications, which are described in chapter 4.7.

4.8.1.2.1 NvM_RpNotifyFinished_Id<BlockName>

A port of type **NvM_RpNotifyFinished_Id** is a Rport of one NVRAM block, which is configured to use Service Ports.

If the SWC setting **Long Service Port Names** is enabled, the name of the service ports is NvM_RpNotifyFinished_Id<BlockName>; if **Long Service Port Names** is disabled, the name is NvM_RpNotifyFinished_Id<BlockId>.

Available in all API Config Classes but **Use Callbacks** must be enabled.

Operation	Notification
JobFinished	SingleBlockCallbackFunction

Table 4-41 Operation of Port prototype NvM_RpNotifyFinished_Id<BlockName>

5 Configuration

5.1 Software Component Template

5.1.1 Generation

The definition of the Provide Ports is described in an XML file. This file describes the NVM as a software component with ports to which other applications can connect. This XML file is always saved consistent to the current ECUC file when the project in DaVinci Configurator is saved. The target directory for SW-C files can be set in the Dpa file. For more information see documentation of DaVinci Configurator.

5.1.2 Import into DaVinci Developer

For further processing the generated software component template file has to be imported into DaVinci Developer. This can be done while a DaVinci-project is open by clicking **File** | **Import XML File...**. Choose the correct file for the import.

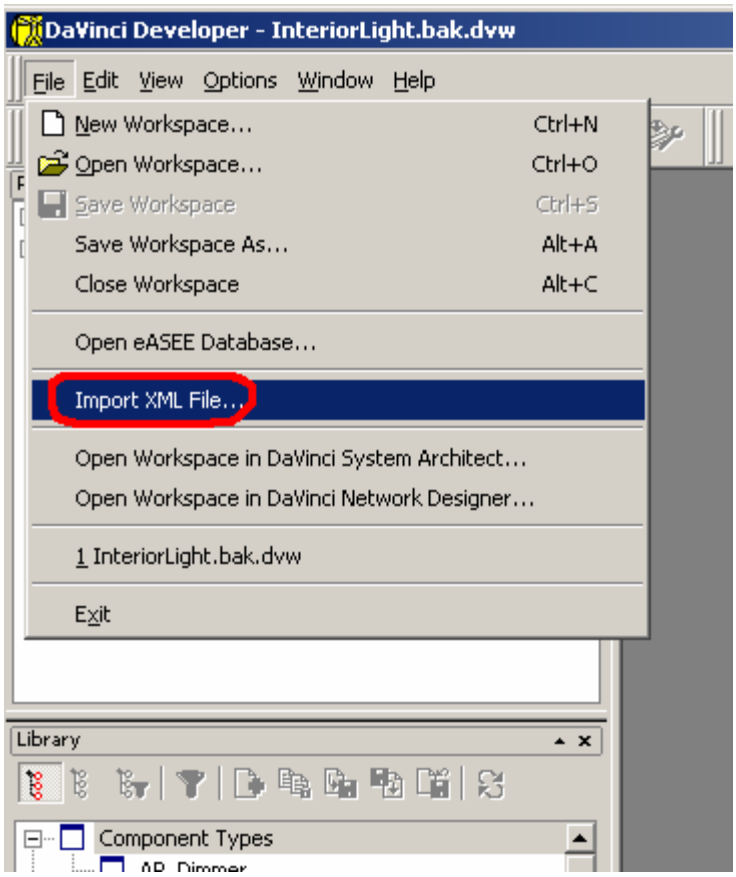

Figure 5-1 Import a new software component into DaVinci Developer

After importing the NVM as software component there is a new component type in the library view available. After double clicking the component NVM, all configured ports are presented.

The DaVinci tool suite lets you design the complete architecture of a car, consisting of several ECUs, each with its own NVM. Therefore it is desirable to import several NVM SW-C descriptions, each containing the description of an NVM to be mapped to a particular ECU. Using the ‘Service Component Name Parameter’ you can give your configurations meaningful unique names. All elements of the SW-C description are unique in this particular

configuration and are prefixed with this parameter's value. However, most elements are common to all SW-C descriptions, or are at least unique to the used configuration (which is also expressed by the elements' names) so that some elements are contained in each different SW-C description. During import, DaVinci will warn you about these doubled elements. You can ignore them (overwrite the existing elements); they are identical.

5.1.3 Dependencies on Configuration of NVM Attributes

The configuration of the NVM attributes (described in chapter 5.1.5) highly influences the resulting SW-C Description. So, the value of the parameter **Service Component Name** influences the names of several elements in the description, especially the name of the **Service Component**. It is also the prefix for several other names that belong to this particular NVM configuration (and the resulting service component).

There is a couple of different port interfaces that will be generated, depending on the particular configuration. Each generated interface that results from a specific configuration has a unique name, i.e. in different SW-C descriptions port interfaces with the same name are compatible; they provide the same operations, each with the same arguments of same type.

5.1.3.1 Naming of Service Port Interfaces

The Service Port Interface provides the prototypes of the elementary block related services of the NVM, such as read data from NV memory, write data to NV memory. It generally contains the string **Service**.

As described above, port interfaces resulting from different configurations, have different names. These names are given according to this scheme:

- Each Interface is prefixed by **NvM**
- **Set Ram Block Status Api**
  If enabled, the interface name contains the string 'SRBS', and it contains the operation SetRamBlockStatus.
- **API Configuration Class**
  The interface name contains a short string that denotes the API configuration class it belongs to: **AC1**, **AC2** or **AC3**. The operations the interface describes in that configuration class are described in Chapter 4.8.1.1.
- **Availability of ROM default data**
  The interface contains the operation RestoreBlockDefaults; it contains the string **Defs**. This interface will be used by all P-Port-Prototypes belonging to a NVRAM block that was configured with ROM default data.
- **Block Management Type DATASET**
  The interface provides the operations GetDataIndex and SetDataIndex. Its name contains **DS**. This interface will be used by all NVRAM blocks of Management Type **DATASET**

The first two possibilities are common within one SW-C Description. Only one combination of them will occur. Unless **API Configuration Class 1** was chosen, Port Interfaces describing any combination of the latter two possibilities may be generated.

5.1.4 Service Port Prototypes

For each active NVRAM block (including the configuration ID block) that was configured with **Use Service Ports** port, prototypes will be generated. The port interfaces they are based on can differ. The interfaces depend on the block's configuration, and hence on the operations that are necessary for current block.

5.1.4.1 Port Prototype Naming

The short name uniquely identifying the prototype is based on the numeric block ID (which, in turn, is derived from the block's position in the configuration) and the port interface **class** it corresponds to.

Each prototype is prefixed by the String **NvM_**; the next substring describes the corresponding port interface, and whether it is a Provide Port ('Pp') or a Require Port ('Rp'):

- **Padmin**
  Linked with port interface **NvMAdministration** (only in **API Configuration Class 3**)
- **PS**
  Linked with Port Interface 'NvMService_AC{1|2|3}[_SRBS][_Defs][_DS]'. The actual interface depends on the possibilities described above.
- **NvM_RpNotifyFinished**
  Linked with Port Interface NvMNotifyJobFinished that describes the interface used by the NVM for **single block job end notification**

If SWC setting **Long Service Port Names** is disabled, each port prototype's name is post fixed by **_Id{BlockId}**. If SWC setting **Long Service Port Names** is enabled, each port prototype's name is post fixed by '_{BlockName}'.

Additionally each port prototype contains a long name as well as a description, which describe it in a better, human readable form. They contain the logical block name, as configured, instead of the block ID, and the used port interface's short name.

5.1.5 Modelling SWC's callback functions

According to AUTOSAR, the prototype of a SingleBlockCallbackFunction (Chapter 4.7.1), differs from that of a RUNNABLE implementing SWC's behavior of that callback. Therefore the prototype describes the RTE function called by NVM.

The prototype of the RUNNABLE, which is actually called by RTE, must match model, i.e. the return type must match the information given in callback's interface description ("Application Errors"). The correct modelling would be "no Application Errors", which requires a RUNNABLE implementation without return type:

```
void <init_cbk_runnable_name>(void)

void <jobend_cbk_runnable_name>( NvM_ServiceIdType ServiceId,
                                 NvM_RequestResultType JobResult)
```

However, DaVinci Developer since Version 3.8 along with MICROSAR RTE version 4.04.00 and later enable NVM to support another different (actually incompatible) function prototype:

```
Std_ReturnType <init_cbk_runnable_name>(void)

Std_ReturnType <jobend_cbk_runnable_name>( NvM_ServiceIdType
                                           ServiceId,
                                           NvM_RequestResultType JobResult)
```

Both implementations require slightly different Interface definitions; they may be adapted using DaVinci Developer. From a modeling point of view, the runnable must be implemented according to the Interface associated with the related Pport-Prototype.

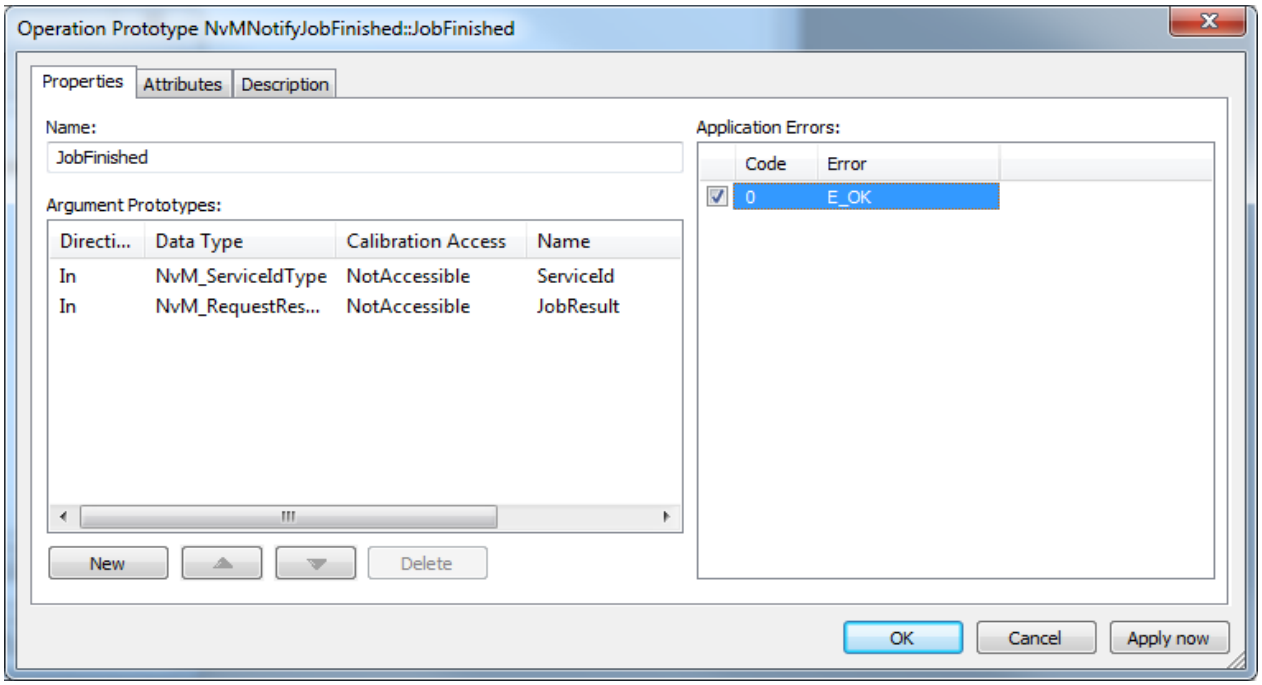

Figure 5-2 A “Single Block Job End Notification” with return type Std_ReturnType

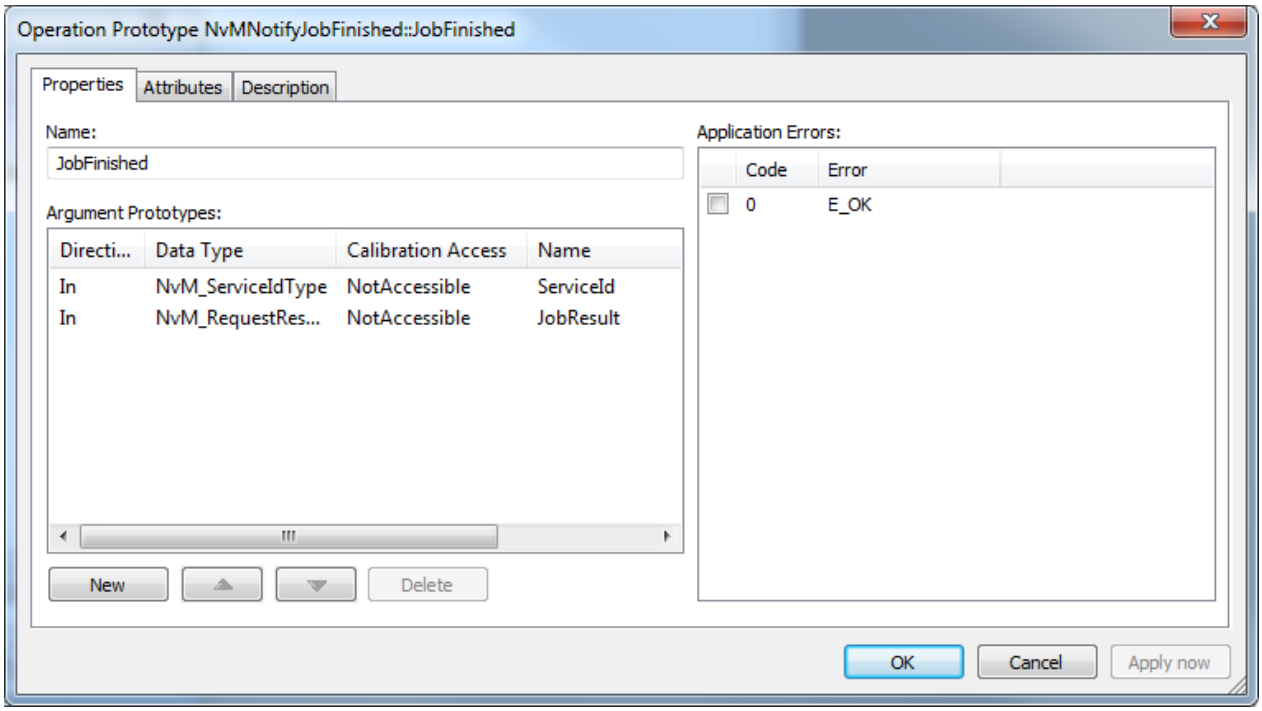

Figure 5-3 A “Single Block Job End Notification” with return type void.

NVM itself provides interfaces as described by Figure 5-3. To make SWCs independent of this definition, they may associate their PPort prototype to their own interface definitions,

according to their (planned) RUNNABLEs' implementations. Of course, the interface must be compatible according to AUTOSAR's rules; which limits possible interface definitions to exactly one of both mentioned here.

5.2 Configuration of NVM Attributes

The NVM attributes can be configured using the DaVinci Configurator. The outputs of the configuration and generation process are the configuration source files.

The description of each used parameter is set in the NvM bswmd file.

Only additional information is given in this chapter.

Caution

Because `sizeof`-operator cannot be used during configuration in production code because sizes also affect lower layers, the exact sizes of your NVRAM blocks, and hence your data structures must be known at configuration time. Therefore you are required to determine these values by yourself. This leads to some significant pitfalls:

- The sizes of basic data types are platform dependent. To handle this problem, you should use only AUTOSAR data types as defined in `Std_Types.h` (respectively `Platform_Types.h`). They are defined to have the same size on all platforms. The enumeration type's size also depends on your platform, the compiler and its options. Be aware of the size the compiler actually chooses. Usually an `enum` equals to an `int` by default, but you can force it to be the smallest possible type (e.g. `char`).
- Be aware of the composition of bit fields. It can be affected by compiler switches.
- The compiler may rearrange members of structures to save memory. The best solution would be to arrange members according to their type manually. The compiler may add unused padding bytes to increase accessibility to the members of a structure. According to the previous fact, you should order your structure's members. Doing so, you should be aware of aligned start addresses for larger integral data types (e.g. `uint16` or `uint32`) according to the CPU's requirements for accessing them.
- As stated above, some compiler switches influence the sizes of data types. Keep in mind that changing these ones may result in changed sizes of your data blocks, leading to a reconfiguration of NVM.

A good way to determine the blocks' sizes is to extract the required information from the linker file or from the generated object.

5.2.1 NvM Block Use Set Ram Block Status

The service NvM_SetRamBlockStatus can be used according to AUTOSAR specification 4.2.2. This allows a per NvBlock configurable API within the component configuration. NvM will reject all requests for the NvM_SetRamBlockStatus API on NvBlocks, which have the NvMBlockUseSetRamBlockStatus configuration parameter disabled.

This feature can only be used, if the global NvMSetRamBlockStatusApi is enabled in configuration.

5.2.2 NvM Invoke Callbacks For Read/Write All

The parameters NvMInvokeCallbacksForReadAll and NvMInvokeCallbacksForWriteAll allow to control whether the configured single block callback shall be invoked during NvM_ReadAll/NvM_WriteAll/NvM_PreReadAll.

Invoked callbacks can be configured via the parameters NvMSingleBlockCallback and/or NvMSingleBlockCallbackExtended.

5.2.3 Select block for Read All and Pre Read All

With the parameters NvMSelectBlockForPreReadAll and NvMSelectBlockForReadAll it is possible to select a block for NvM_PreReadAll and NvM_ReadAll. A block can be selected for both. In this case the block is only processed during NvM_PreReadAll. This behavior avoids validation errors if the user wants that a block is read during NvM_PreReadAll but the component which uses this block, validates that it is selected for NvM_ReadAll.

5.2.4 NvM Using MAC as data integrity mechanism

To enable MAC as data integrity mechanism for a block, three parameters must be configured.

NvMCsmMacGenerationJobReference and NvMCsmMacVerificationJobReference both must reference the required CSM jobs to generate and verify the MACs for the block. The referenced jobs must themselves reference the correct CsmPrimitives (CsmMacGenerate and CsmMacVerify) where the processing must be set to CSM_SYNCHRONOUS (the current NvM implementation only support synchronous CSM jobs).

The third parameter, NvMCsmMacSize, specifies the actual size of the MAC that is generated from the block data.

Caution
It is not allowed to use the MAC data integrity mechanism for safety related data.

6 AUTOSAR Standard Compliance

6.1 Deviations

- In contrast to AUTOSAR most configuration parameters are link-time parameters.
- Saving RAM CRC of current block is configuration dependent. Either it is saved behind the block's data or it is saved internally by NVM in an own variable.
- Unified handling of ROM defaults among all block management types is processed. Rom defaults handling of blocks of type dataset is just like the handling of blocks of the other management types.
- NVM is able to provide the Config Id's RAM block on its own.
- `NvM_WriteAll()` does not write unchanged data, even if this would repair (redundant) NV data.
- Attempts to write to a locked block (`NvM_SetBlockLockStatus`) are explicitly not treated as Development Error; error NVM_E_BLOCK_LOCKED is not defined.
- `NvM_SetBlockLockStatus` is allowed for pending Blocks; no related Development Error Check will be performed.
- Block CRC type CRC8 is not supported.
- Write retries can only be globally configured, rather than individually per NVRAM Block
- Calls to Explicit Synchronization callbacks (see chapter 2.4.20) cannot be limited by configuration. Rather those functions are expected to succeed within few attempts.

6.2 Additions/ Extensions

6.2.1 Parameter Checking

The internal parameter checks of the API functions can be en-/disabled separately. The AUTOSAR standard requires en-/disabling of the complete parameter checking only. For details see chapter 2.5.1.1.

6.2.2 Concurrent access to NV data

NVM provides for DCM possibility to access NV data concurrently with NVM application. (see chapter 2.4.19)

6.2.3 RAM-/ROM Block Size checks

NVM can be configured to check all RAM and ROM blocks' lengths against corresponding NV Block lengths, using `sizeof` operator; see chapter 2.5.3.

6.2.4 Calculated CRC value does not depend on number of calculation steps

Due to the specified CRC32 algorithm, and missing further requirements on NVM's CRC calculation, a calculated CRC32 value depends on the number of necessary calculation steps (defined by block length and parameter **CRC Bytes per Cycle**). Unless the CRC can be calculated with one step (i.e. the block is small enough), the CRC32 value will not match the value resulting from calling the CRC32 library function once for the whole block.

The reason is the negation of the result, as specified for CRC32 (which in turn belongs to standard/widely used **Ethernet CRC**). This behavior introduces some drawbacks on NVM, especially:

- Changing parameter **CRC Bytes per Cycle** (for run-time optimization), in an existing (already flashed) project. Data blocks with CRC32 could not be read after the update.
- CRC32 values cannot be verified outside NVM (e.g. for testing purposes), without knowing the configuration – each single step would have to be reproduced.
- Valid data blocks along with their CRC32 cannot be pre-defined using standard CRC algorithms.

NVM circumvents these restrictions by reverting the final negation of each single CRC32 calculation step, except the last one. This (quite simple) measure guarantees that the CRC value does NOT depend on the number of calculation steps, as it is originally guaranteed for CRC16 (since it will not be inverted by the CRC library).

6.2.5 Support of AUTOSAR 4.4.0 SingleBlockCallback / MultiBlockCallback

With AUTOSAR version 4.4.0 the interfaces of `SingleBlockCallback` and `MultiBlockCallback` was changed in a way that backward compatibility was not given anymore.

To support legacy systems but also systems requiring the new interfaces the configuration parameter `NvM/NvMCommon/NvMUseAsr440CallbackInterface` was added. When enabling this parameter, the callbacks will be invoked with dedicated types instead of the former `ServiceId`.

6.2.5.1 Behavioral change

The ServiceId mapping was not changed exactly 1:1 when the change was introduced. Instead of using the `SingleBlockCallback` with `NVM_READ_ALL`, `NVM_READ_ALL_BLOCK` will be used to notify that a block was read.

Caution
Please ensure that the SWC components and BSW components use the correct `NvM` literals defined in NvM_Types.h when using the callbacks as the behavior has changed.

6.2.5.2 NvM_BlockRequestType extension

Some users of `NvM` are also interested in notification callbacks when `NvM_WriteAll()` is executed. To enable the further usage of the callbacks, the block request type was extended to also support `NVM_WRITE_ALL_BLOCK` additionally.

6.3 Limitations

There are no limitations.

7 Cybersecurity

This chapter describes relevant information for a secure integration and configuration, so-called Cybersecurity Manual Instructions (CMI), of this component to fulfil identified Technical Cybersecurity Requirements (TCR). Additionally, functional cybersecurity dependencies to other components are described.

In order to fulfil TCR-20 (Protect and verify the integrity and authenticity of non-volatile data), CMI-NvM-TransformationCallbacks-DataIntegrityAndAuthenticitiy must be considered.

Caution
Please note that it is not possible to use the NvM internal MAC generation and verification described in Chapter 2.4.5.5.2 to fulfil TCR-20, because this mechanism does not allow to include the block identification for protection of the data block's authenticity.

The NvM provides two alternatives to fulfil TCR-21 (Protect the confidentiality of non-volatile data):

a) by configuring a CSM job, or

b) by implementing transformation callbacks.

If approach a) is used, only CMI-NvM-Csm-DataConfidentiality must be considered. If approach b) is used, only CMI-NvM-TransformationCallbacks-DataConfidentiality must be considered.

7.1 Configuration

7.1.1 CMI-NvM-Csm-DataConfidentiality

Technical Cybersecurity Requirements: TCR-21

Instruction:

The user of MICROSAR Classic shall configure a CSM encryption/decryption job to protect the confidentiality of NV data of cybersecurity-relevant blocks.

For more information, see Chapter 2.4.23.

Rationale:

The NvM is working with plain payload. It is necessary to encrypt and decrypt the data to keep it confidential.

7.2 Runtime Interfaces: Application

7.2.1 CMI-NvM-TransformationCallbacks-DataConfidentiality

Technical Cybersecurity Requirements: TCR-21

Instruction:

The user of MICROSAR Classic shall provide a mechanism to protect confidentiality of NV data via the callbacks as described in Chapters 4.7.8 and 4.7.9.

Rationale:

The NvM is working with plain payload. It is necessary to encrypt and decrypt the data to keep it confidential.

7.2.2 CMI-NvM-TransformationCallbacks-DataIntegrityAndAuthenticitiy

Technical Cybersecurity Requirements: TCR-20

Instruction:

The user of MICROSAR Classic shall provide a mechanism to protect data integrity and authenticity of NV data via the callbacks as described in Chapters 4.7.8 and 4.7.9.

Rationale:

To ensure integrity and authenticity of block data, an appropriate data integrity record (e.g., a cryptographic MAC) must be generated and added to the data and verified when reading block data. The block identification must also be included during generation and verification of the data integrity record.

7.3 Runtime Interfaces: BSW

TCR	Depends on Component(s)	Comment
TCR-21	CSM	Needed for encryption/decryption.

Table 7-1 Cybersecurity-relevant Dependencies for Runtime Interfaces

8 Glossary and Abbreviations

8.1 Glossary

Term	Description
DaVinci Configurator Pro	Configuration and generation tool for MICROSAR.
Primary NV Block	The first NV block of an NVRAM Block of type Redundant. During reads, this block will always be tried first. During writes it will be preferred, unless only secondary is defective.
Secondary NV Block	The second NV block of an NVRAM Block of type Redundant. During reads and this block will be accessed second; if primary is defective.

Table 8-1 Glossary

8.2 Abbreviations

Abbreviation	Description
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BSW	Basis Software
CRC	Cyclic Redundancy Check
CSM	Crypto Service manager
DCM	Diagnostic Communication Manager
DEM	Diagnostic Event Manager
DET	Development Error Tracer
DPA	DaVinci Project Assistant
EA	EEPROM Abstraction Module
ECU	Electronic Control Unit
ECUC	ECU Configuration
ECUM	ECU State Manager
EEP	EEPROM Driver
EEPROM	Electrically Erasable Programmable Read Only Memory
FEE	Flash EEPROM Emulation Module
FIFO	First In First Out
FLS	Flash Driver
GCE	Generic Configuration Editor – generic tool for editing AUTOSAR configuration files. In DaVinci Configurator, the view can be switch to **Generic Editor**.
HIS	Hersteller Initiative Software
ISR	Interrupt Service Routine
MAC	Message Authentication Code
MemHwA	Memory Hardware Abstraction Layer

MEMIF	Memory Abstraction Interface Module
MICROSAR	Microcontroller Open System Architecture (the Vector AUTOSAR solution)
NVM	NVRAM Manager
NV, NVRAM	Non Volatile Random Access Memory
OS	Operating System
PIM	Per Instance Memory: Memory (RAM) to be used by an instance of an Software Component, provide by RTE. It may also be used as permanent or temporary RAM block. Such a memory need is usually modeled using a tool like DaVinci Developer.
PPort	Provide Port
RAM	Random Access Memory
PRAM	Permanent Random Access Memory
ROM	Read Only Memory
RPort	Require Port
RTE	Runtime Environment
SRS	Software Requirement Specification
SWC	Software Component
SWS	Software Specification
DataIntegrityRecord	Abstract naming for any Data Integrity mechanism that can be configured, see Data Integrity)

Table 8-2 Abbreviations

9 Contact

Visit our website for more information on

- News
- Products
- Demo software
- Support
- Training data
- Addresses

www.vector.com