

MICROSAR NVM

Technical Reference

Version 5.06.00

Authors	Christian Kaiser, Tomas Ondrovic
Status	Released

1 Document Information

1.1 History

Author	Date	Version	Remarks
Christian Kaiser	2007-08-20	1.4	AUTOSAR 2.1, updated for EAD3.1 usage, conversion to new template
Christian Kaiser	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
Christian Kaiser	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()' added. 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.
Christian Kaiser	2010-01-25	3.04.01	ESCAN00039648 – Rebuilt document; made hyperlinks working. Updated Logo; No changes in content.
Christian Kaiser	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
Christian Kaiser	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, Christian Kaiser	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
Christian Kaiser	2014-06-17	5.02.00	Internal release; no changes
Christian Kaiser	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
Christian Kaiser	2015-01-07	5.02.02	Chapters 6.4.8, 8.1 NvM_SetBlockLockStatus – emphasized deviation from AUTOSAR.
Tomas Ondrovic	2015-02-02	5.03.00	Chapter 4.2.1, 4.5.1 – removed RAM and ROM block length DET check Chapter 4.5.3 – describes the new compile time RAM and ROM block length checks Chapter 4.1.1.1– created to describe feature Block Id check Chapter 6.4.20 – describes the new feature “Repair Redundant Blocks”
Tomas Ondrovic	2015-09-28	5.03.01	Only improvements
Tomas Ondrovic	2015-11-25	5.0400	Added chapter 4.1.2 and updated chapter 4.5.3
Tomas Ondrovic	2016-02-11	5.05.00	Removed the possibility to configure dev error hook, include file and reporting
Tomas Ondrovic	2016-04-14	5.06.00	ESCAN00088791: updated function signatures to AUTOSAR4 in chapter 6.4 FEAT-1888: described changed callback invoking during ReadAll in chapters 4.4.8 and 4.4.13 Added new chapter 5.2 with critical section list.

			ESCAN00091004: extended chapter 4.2
--	--	--	-------------------------------------

Table 1-1 History of the document

1.2 Reference Documents

No.	Title	Version
[1]	AUTOSAR_SWS_NVRAMManager.pdf	V 2.2.0
[2]	AUTOSAR_SWS_DET.pdf	V 2.2.0
[3]	AUTOSAR_SWS_DEM.pdf	V 2.2.1
[4]	AUTOSAR_BasicSoftwareModules.pdf	V 1.2.0

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

1	Document Information	2
1.1	History	2
1.2	Reference Documents	6
2	Component History	13
3	Introduction.....	15
3.1	Architecture Overview	16
4	Functional Description	18
4.1	Features	18
4.1.1	Safety Features.....	19
4.1.2	Automatic Block Length	19
4.2	Initialization	19
4.2.1	Start-up	20
4.2.2	Initialization of the Data Blocks	20
4.3	States	21
4.4	Main Functions	21
4.4.1	Hardware Independence	21
4.4.2	Synchronous Requests	21
4.4.3	Asynchronous Requests	21
4.4.4	API Configuration Classes and additional API Services	22
4.4.5	Block Handling	23
4.4.6	Prioritized or non-prioritized Queuing of asynchronous Requests	27
4.4.7	Asynchronous Job-End Polling	27
4.4.8	Single Block Job End Notifications	27
4.4.9	Immediate Priority Jobs and Cancellation of current Jobs	28
4.4.10	Asynchronous CRC Calculation	28
4.4.11	Write Protection	29
4.4.12	Erase and Invalidate	29
4.4.13	Init Block Callbacks	29
4.4.14	Define Locking/ Unlocking Services	30
4.4.15	Interrupts.....	30
4.4.16	Data Corruption.....	30
4.4.17	Concurrent access to NV data for DCM	30
4.4.18	Explicit synchronization mechanism between application and NVM .	31
4.5	Error Handling.....	32
4.5.1	Development Error Reporting.....	32
4.5.2	Production Code Error Reporting	35

4.5.3	Compile-time Block Length Checks.....	35
5	Integration.....	37
5.1	Scope of Delivery.....	37
5.1.1	Static Files	37
5.1.2	Dynamic Files	37
5.2	Critical Sections	38
5.3	Include Structure.....	38
5.4	Compiler Abstraction and Memory Mapping.....	38
5.5	Dependencies on SW Modules	40
5.5.1	OSEK / AUTOSAR OS	40
5.5.2	DEM.....	41
5.5.3	DET	41
5.5.4	MEMIF	41
5.5.5	CRC Library	41
5.5.6	Callback Functions.....	41
5.5.7	RTE	41
5.5.8	BSWM.....	41
5.6	Integration Steps.....	42
5.7	Estimating Resource Consumption	43
5.7.1	RAM Usage	43
5.7.2	ROM Usage	43
5.7.3	NV Usage	44
5.8	How-To: Integrate NVM with AUTOSAR3 SWC's	44
5.8.1	NVM's provided Interfaces/Ports	44
5.8.2	Callbacks (Ports provided by client SWCs)	45
5.8.3	Request Result Types	45
6	API Description.....	46
6.1	Interfaces Overview	46
6.2	Type Definitions	46
6.3	Global API Constants	48
6.4	Services provided by NVM.....	48
6.4.1	NvM_Init.....	48
6.4.2	NvM_SetDataIndex.....	48
6.4.3	NvM_GetDataIndex.....	49
6.4.4	NvM_SetBlockProtection	50
6.4.5	NvM_GetErrorStatus.....	51
6.4.6	NvM_GetVersionInfo	51
6.4.7	NvM_SetRamBlockStatus	52
6.4.8	NvM_SetBlockLockStatus	53

6.4.9	NvM_MainFunction	54
6.4.10	NvM_ReadBlock	54
6.4.11	NvM_WriteBlock	55
6.4.12	NvM_RestoreBlockDefaults	56
6.4.13	NvM_EraseNvBlock	57
6.4.14	NvM_InvalidateNvBlock	58
6.4.15	NvM_ReadAll	59
6.4.16	NvM_WriteAll	60
6.4.17	NvM_CancelWriteAll	61
6.4.18	NvM_KillWriteAll	61
6.4.19	NvM_CancelJobs	62
6.4.20	NvM_RepairRedundantBlocks	62
6.5	Services used by NVM.....	63
6.6	Callback Functions.....	64
6.6.1	NvM_JobEndNotification	64
6.6.2	NvM_JobErrorNotification	65
6.7	Configurable Interfaces	65
6.7.1	SingleBlockCallbackFunction	65
6.7.2	MultiBlockCallbackFunction	66
6.7.3	InitBlockCallbackFunction	66
6.7.4	Callback function for RAM to NvM copy	67
6.7.5	Callback function for NvM to RAM copy	68
6.8	Service Ports	68
6.8.1	Client Server Interface	68
7	Configuration.....	71
7.1	Software Component Template	71
7.1.1	Generation	71
7.1.2	Import into DaVinci Developer.....	71
7.1.3	Dependencies on Configuration of NVM Attributes.....	72
7.1.4	Service Port Prototypes	73
7.1.5	Modelling SWC's callback functions.....	73
7.2	Configuration of NVM Attributes	75
8	AUTOSAR Standard Compliance.....	77
8.1	Deviations	77
8.2	Additions/ Extensions.....	77
8.2.1	Parameter Checking	77
8.2.2	Concurrent access to NV data	77
8.2.3	RAM-/ROM Block Size checks.....	77

8.2.4	Calculated CRC value does not depend on number of calculation steps	77
8.3	Limitations.....	78
9	Glossary and Abbreviations	79
9.1	Glossary	79
9.2	Abbreviations	79
10	Contact.....	81

Illustrations

Figure 3-1	AUTOSAR 4.x Architecture Overview	16
Figure 3-2	Interfaces to adjacent modules of the NVM	17
Figure 5-1	The file structure of the NVM sections module	38
Figure 7-1	Import a new software component into DaVinci Developer	71
Figure 7-2 A	“Single Block Job End Notification” with return type Std_ReturnType	74
Figure 7-3 A	“Single Block Job End Notification” with return type void.	75

Tables

Table 1-1	History of the document.....	6
Table 1-2	Reference documents.....	6
Table 2-1	Component history.....	14
Table 4-1	Supported SWS features	18
Table 4-2	Not supported SWS features	18
Table 4-3	Block concept	25
Table 4-4	Mapping of service IDs to services	33
Table 4-5	Errors reported to DET	34
Table 4-6	Development Error Checking: Assignment of checks to services	34
Table 4-7	Errors reported to DEM.....	35
Table 5-1	Static files	37
Table 5-2	Generated files	37
Table 5-3	Compiler abstraction and memory mapping.....	39
Table 6-1	Type definitions.....	48
Table 6-2	NvM_Init	48
Table 6-3	NvM_SetDataIndex.....	49
Table 6-4	NvM_GetDataIndex	50
Table 6-5	NvM_SetBlockProtection	50
Table 6-6	NvM_GetErrorStatus	51
Table 6-7	NvM_GetVersionInfo.....	52
Table 6-8	NvM_SetRamBlockStatus.....	52
Table 6-9	NvM_SetBlockLockStatus.....	53
Table 6-10	NvM_MainFunction.....	54
Table 6-11	NvM_ReadBlock	55
Table 6-12	NvM_WriteBlock	56
Table 6-13	NvM_RestoreBlockDefaults	57
Table 6-14	NvM_EraseNvBlock.....	58
Table 6-15	NvM_InvalidateNvBlock	59
Table 6-16	NvM_ReadAll.....	60
Table 6-17	NvM_WriteAll	61
Table 6-18	NvM_CancelWriteAll	61
Table 6-19	NvM_KillWriteAll	62
Table 6-20	NvM_CancelJobs	62
Table 6-21	NvM_RepairRedundantBlocks	63
Table 6-22	Services used by the NVM.....	64
Table 6-23	NvM_JobEndNotification	64
Table 6-24	NvM_JobErrorNotification	65
Table 6-25	SingleBlockCallbackFunction.....	66
Table 6-26	MultiBlockCallbackFunction	66
Table 6-27	InitBlockCallbackFunction.....	67
Table 6-28	Callback function for RAM to NvM copy.....	68

Table 6-29	Callback function for NvM to RAM copy.....	68
Table 6-30	Operations of Port Prototype Padmin_<BlockName>	69
Table 6-31	Operations of Port Prototype PS_<BlockName>.....	69
Table 6-32	Operation of Port prototype NvM_RpNotifyFinished_Id<BlockName>	70
Table 9-1	Glossary	79
Table 9-2	Abbreviations.....	80

2 Component History

Component Version (Implementation)	New Features
5.05.xx	Reworked job end and init block callback invocation during ReadAll for blocks with service ports
5.03.xx	Specific development errors cannot be switched on/off, only global development error mode can be enabled/disabled
5.02.xx	Added RepairRedundantBlocks functionality SafeBSW
5.01.xx	Interaction with BswM added. Defined Block Write Order (descending IDs) during write all
5.xx.xx	AUTOSAR4.0. <ul style="list-style-type: none"> • Changed API (return types) • New service: <code>NvM_CancelJobs</code> • New DEM Errors • "Write Block during WriteAll" configurable • Explicit Synchronization Mechanism
4.xx.xx	Skipped/Reserved.
3.09.xx	Removed AUTOSAR Version Check for DEM <code>NvM_Mainfunction</code> runnable is always generated into SWC description Generation of SWC Interface Names according to AUTOSAR
3.08.xx	NVM provides information about error codes for MICROSAR Dem to automate configuration.
3.07.xx	Service <code>NvM_KillWriteAll</code> added. Significant changes in internal handling (CRC/internal buffer)
3.06.xx	Never released; no new features.
3.05.xx	Calculated CRC32 value does not depend anymore on configuration of parameter <code>NvmCrcNumOfBytes</code> . Added RAM and ROM block size checks: The NVM can be configured to check each RAM block's and/or each ROM block's size against the configured NV block length, considering CRC setting, internal buffering, etc.
3.04.xx	Crc Handling is configurable: Either the internal buffer, available since component version 3.02, is used or Crc is stored at the end of RAM Block.
3.03.xx	At processing a redundant NVRAM Block NVM determines an appropriate write order, depending on the NV Block's current state/content. A defect NV block is written in preference to a valid one. NVM provides possibility for DCM to access NV data concurrently with NVM's applications.
3.02.xx	Update to AUTOSAR 3 specification. Additional API <code>NvM_SetBlockLockStatus</code> . Storing each NVRAM block's CRC internally: RAM Blocks provided by the application don't have to allocate additional space for CRC.

Component Version (Implementation)	New Features
	Configurability, whether the NVM shall create the RAM Block associated with the ConfigID NVRAM Block on its own, or the user creates the RAM block.

Table 2-1 Component history

3 Introduction

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

Supported AUTOSAR Release*:	4	
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. [4])

* For the precise AUTOSAR Release 3.x please see the release specific documentation.

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!

3.1 Architecture Overview

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

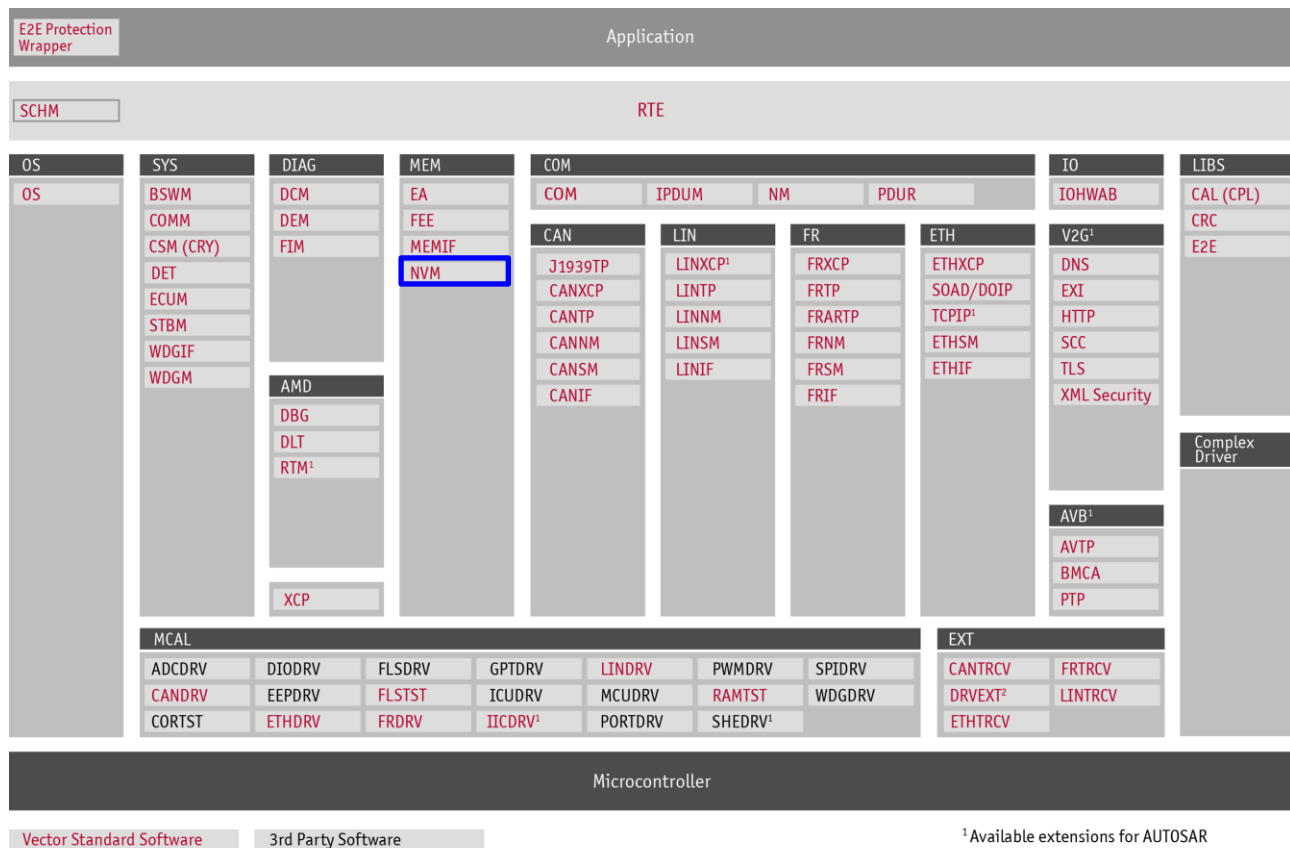


Figure 3-1 AUTOSAR 4.x Architecture Overview

The next figure shows the interfaces to adjacent modules of the NVM. These interfaces are described in chapter 6.

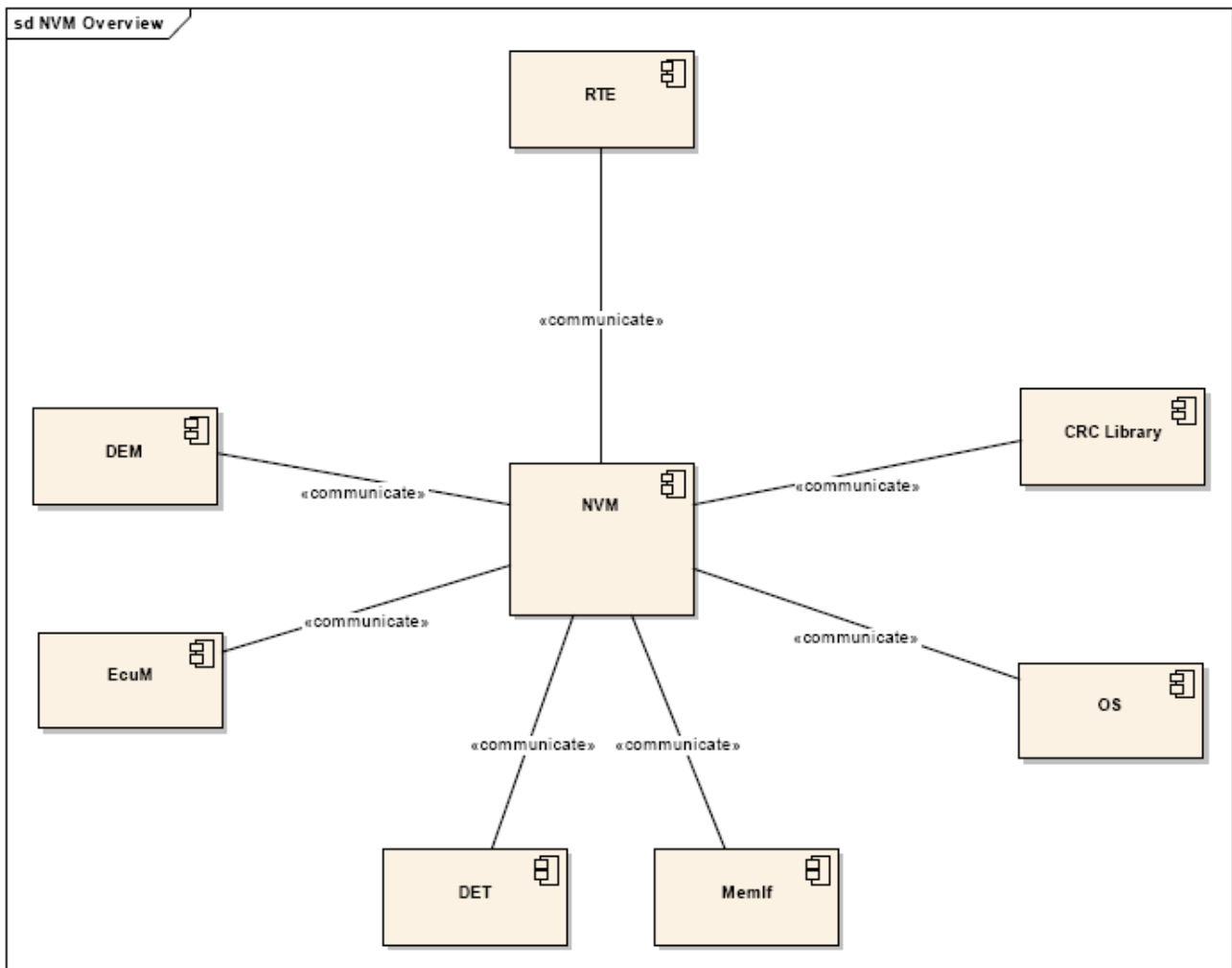


Figure 3-2 Interfaces to adjacent modules of the NVM

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 6.8 and are defined in [1].

4 Functional Description

4.1 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 8.

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 4-1 Supported SWS features

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 4-2 Not supported SWS features

4.1.1 Safety Features

4.1.1.1 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 than 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.

4.1.2 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 4.5.3) 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.

4.2 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 4.2.1)
2. `NvM_ReadAll()` (see 4.2.2)

Independently 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()`.

4.2.1 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.

4.2.2 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.

Block 1 (the configuration ID) has a special role. It is stored in NV memory and also as a constant (`NvM_CompiledConfigId_t`) that is externally visible and link-time configurable. During `NvM_ReadAll()` the NV value of block 1 is compared against the constant `NvM_CompiledConfigId_t`. In case of a match all NV blocks are presumed to be valid and NVM tries to read them from NV memory. In case of a mismatch or if the configuration ID cannot be read the system behaves as following:

- > If the configuration switch **Dynamic Configuration Handling** is **OFF**, the mismatch is ignored. It will be tried to read all blocks from NV memory (also called 'normal runtime preparation').
- > If the **Dynamic Configuration Handling** is **ON**, the normal runtime preparation is processed for all blocks having been configured with the option 'Resistant to Changed SW'. For all other blocks an 'extended runtime preparation' will take place.

- > 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.

4.3 States

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

4.4 Main Functions

4.4.1 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.

4.4.2 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()`

4.4.3 Asynchronous Requests

Following NVM API services initiate asynchronous requests:

- > `NvM_ReadBlock()`
- > `NvM_WriteBlock()`
- > `NvM_RestoreBlockDefaults()`

```
> NvM_EraseNvBlock()  
> NvM_InvalidateNvBlock()  
> NvM_SetRamBlockStatus() (for CRC protected blocks)  
> NvM_ReadAll()  
> NvM_WriteAll()  
> NvM_CancelWriteAll()  
> NvM_CancelJobs()
```

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.

4.4.4 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_GetErrorStatus()  
> NvM_SetRamBlockStatus()  
> NvM_ReadAll()  
> NvM_WriteAll()  
> NvM_CancelWriteAll()
```

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_SetDataIndex()`
- > `NvM_GetDataIndex()`
- > `NvM_ReadBlock()`
- > `NvM_WriteBlock()`
- > `NvM_RestoreBlockDefaults()`
- > `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()` and `NvM_GetVersionInfo()` can be enabled/disabled additionally via the configuration tool. The function `NvM_SetBlockLockStatus()` is always available independent of API configuration class.

4.4.5 Block Handling

4.4.5.1 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).

4.4.5.2 Different Types of NV Blocks

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

4.4.5.2.1 Native Blocks

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

4.4.5.2.2 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 CRC usage for Redundant Blocks, because CRC provides 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 it detects a defective NV Block, it is written in preference to a valid NV Block. If writing to one single NV Block failed, the NVM reports the error `NVM_E_REQ_FAILED` (see chapter 4.5.2) to the DEM. If writing to primary NV block failed, NVM ends the request always with a negative job result. If the primary NV block was written successfully, the request always ends with a positive job result, even when the secondary NV block failed.

**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.

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.

4.4.5.2.3 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 4-3 Block concept

4.4.5.3 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.

4.4.5.4 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 4.4.13) 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 explicit (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 CRC error or because of a failure reported by the underlying MemHwA via MEMIF.

4.4.5.5 Checksum

For each block an optional 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 **Internal Buffer for CRC Handling** is disabled, Storage for CRC 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 CRC) into an internal buffer, instead of directly passing them down to lower layers. Here, data gets appended with CRC, in order to keep both within one NV block, which requires NVM to pass both down with one single write request.

If “Calc RAM CRC” 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.

4.4.6 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.

4.4.7 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.

4.4.8 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.

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



Note

There are two different exceptions where the NvM does not invoke the job end notification:

- > During `NvM_WriteAll()` the single block job end notification won't be called
- > During `NvM_ReadAll()` the single block job end notification won't be called, if the block is configured with enabled `NvMUseServicePorts`. This will be done because during the `NvM_ReadAll()` the RTE is not initiated yet and callback invocations could lead to DET error. For all blocks with disabled `NvMUseServicePorts` the callbacks will be invoked during `NvM_ReadAll()` without restrictions.

4.4.9 Immediate Priority Jobs and Cancellation of current Jobs

If job prioritization is selected, blocks of different priority exist. A new queued, higher priority job, (e.g. priority 5) does not cancel/suspend a lower prioritized job (e.g. priority 10) if this job is already processed.

The only exceptions for this are immediate priority jobs (priority 0) which can suspend a running job that priority is less. The suspended job will be restarted after all jobs with higher priority are finished.

**Caution**

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

4.4.10 Asynchronous CRC Calculation

The (re-)calculation of a block's CRC is done asynchronously by the `NvM_MainFunction()`. A CRC protected block's CRC value is calculated every time the block shall be written to NV memory. If a block is read from NV memory the CRC value is recalculated and compared to the one that was just read from NV memory. If configuration option 'Calculate RAM CRC' was enabled for a block, its recently calculated CRC value will be stored in RAM for later use.

If `NvM_SetRamBlockStatus(TRUE)` is called, the re-calculation of the CRC value over the RAM block's data will also be initiated, if 'Calculate RAM CRC' was enabled for this block.

**Note**

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 attempts so during `NvM_ReadAll` processing for all NVRAM blocks having 'Read during ReadAll' and 'Calc RAM CRC' 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 CRC calculation is done in the cyclically called service `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.

**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.

For CRC32 this is a feature in NvM, beyond the requirements of AUTOSAR.

4.4.11 Write Protection

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

A block can also be configured to be written once. The write protection of such a 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()`.

**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.

4.4.12 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.

4.4.13 Init Block Callbacks

For any block ROM defaults (see 4.4.5.4) 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.
- > During `NvM_ReadAll()` the initialization callback won't be called, if the block is configured with enabled `NvMUseServicePorts`. This will be done because during the `NvM_ReadAll()` the RTE is not initialed yet and callback invocations could lead to DET error. For all blocks with disabled `NvMUseServicePorts` the callbacks will be invoked during `NvM_ReadAll()` without restrictions.

4.4.14 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').

4.4.15 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.

4.4.16 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.

4.4.17 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. Aliases are neither read at start-up (during `NvM_ReadAll` processing) nor written at shut-down (during `NvM_WriteAll` processing). In fact, they are treated as NVRAM Blocks without permanent RAM block. Consequently, explicit read or write requests must supply a reference to a temporary RAM block

For accessing the alias of a NVRAM block, NVM provides the global macro `NvM_GetDcmBlockId(<BlockId>)`. The macro expects the `BlockId` of the original NVRAM block as parameter and returns the block's alias of type `NvM_BlockIdType`. Only one asynchronous request of one alias can be queued at a time. Otherwise the asynchronous API returns with `E_NOT_OK`, which indicates that the request has not been accepted, because of the block pending state check.

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.

**Note**

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

**Caution**

DCM should lock the block using `NvM_SetBlockLockStatus` (see chapter 6.4.8) 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 (`NvM_ReadAll` processing).

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

4.4.18 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 Crc calculation (see chapter 4.4.5.1). The size of the internal buffer is the size of the biggest configured block plus configured Crc bytes.

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.

4.4.18.1 Explicit synchronization mechanism during write requests

After application issued `NvM_WriteBlock`, 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.

4.4.18.2 Explicit synchronization mechanism during read requests

After application issued `NvM_ReadBlock`, 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.

4.5 Error Handling

4.5.1 Development Error Reporting

By default, development errors are reported to the DET using the service `Det_ReportError()` as specified in [2], 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 3].

The reported service IDs identify the services which are described in 6.4. 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()
0x06	NvM_ReadBlock()
0x07	NvM_WriteBlock()
0x08	NvM_RestoreBlockDefaults()
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()

Table 4-4 Mapping of service IDs to services

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 blocks. If these functions are called with any other block type, this error code is produced. NvM_RestoreBlockDefaults() is restricted to blocks configured with ROM defaults or an init callback.
0x0C NVM_E_PARAM_BLOCK_DATA	NvM_SetDataIndex() may check the range of the

Error Code		Description
	IDX	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 4-5 Errors reported to DET

4.5.1.1 Parameter Checking

AUTOSAR requires that API functions check the validity of their parameters. The checks in Table 4-6 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_WriteBlock()		■			■	■	■
NvM_RestoreBlockDefaults()		■	■		■		■
NvM_EraseNvBlock()		■			■	■	
NvM_CancelWriteAll()		■					
NvM_InvalidateNvBlock()		■			■	■	
NvM_ReadAll()		■		■			
NvM_WriteAll()		■		■			
NvM_MainFunction()		■					
NvM_CancelJobs()		■			■		

Table 4-6 Development Error Checking: Assignment of checks to services

4.5.2 Production Code Error Reporting

Production code related errors are reported by default to the DEM using the service `Dem_ReportErrorStatus()` as specified in [3].

However, the service to be used (and the appropriate include file) for error reporting may be configured.

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

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 4-7 Errors reported to DEM

According to AUTOSAR component specific DEM errors must be configured in DEM. If MICROSAR DEM is used, production code errors are automatically inserted into DEM configuration.

If another module is used for production code error reporting, the function name for reporting the error can be configured by the integrator, but must have the same signature as the service `Dem_ReportErrorStatus()`.

Both error codes shall have a value of type `Dem_EventIdType` (an integer type). It must be assured, that these two error codes as well as the type are **published** to the NVM via the user-specified include file.

4.5.3 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 4.1.2): 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)

5 Integration

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

5.1 Scope of Delivery

The delivery of the NVM contains the files which are described in the chapters 5.1.1 and 5.1.2:

5.1.1 Static Files

File Name	Description
NvM.h	This file must not be modified by user. Defines the interface of NVM. Only this file shall be included by the application.
NvM_Cbk.h	This file must not be modified by user. Contains the declarations of the callback functions being invoked by EEPROM driver
NvM_Types.h	This file must not be modified by user. Defines general types used by NVM.
NvM.c / NvM.lib/NvM.a	This file must not be modified by user. Implementation of NVM, delivered as object library.
NvM_Act / NvM_Crc / NvM_JobProc / NvM_Qry / NvM_Queue.c *.h	These are files for internal use of the NvM. If NVM is delivered as object then this parts are content of NvM.lib.

Table 5-1 Static files

5.1.2 Dynamic Files

The dynamic files are generated by the configuration tool DaVinci Configurator. Do not modify them manually.

File Name	Description
NvM_Cfg.c	It contains configuration parameters of NVM which can be modified after compilation of NvM.c.
NvM_Cfg.h	Contains public configuration parameters of NVM. They are (or might be) also important to NvM's user(s), or they may affect NvM's API It contains also public types and symbol declarations to be used by NVM as well as its user(s).
NvM_PrivateCfg.h	Contains parameters as well as type and symbol declarations, which are private to the NvM, i.e. they only affect internal behavior. This file is intended to be included only by NvM's sources.

Table 5-2 Generated files

5.2 Critical Sections

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

- > NvM_NVM_EXCLUSIVE_AREA_0

5.3 Include Structure

The following figure illustrates the hierarchy of included files. It also shows that Std_Types.h and Nvm.h must be included by the application.

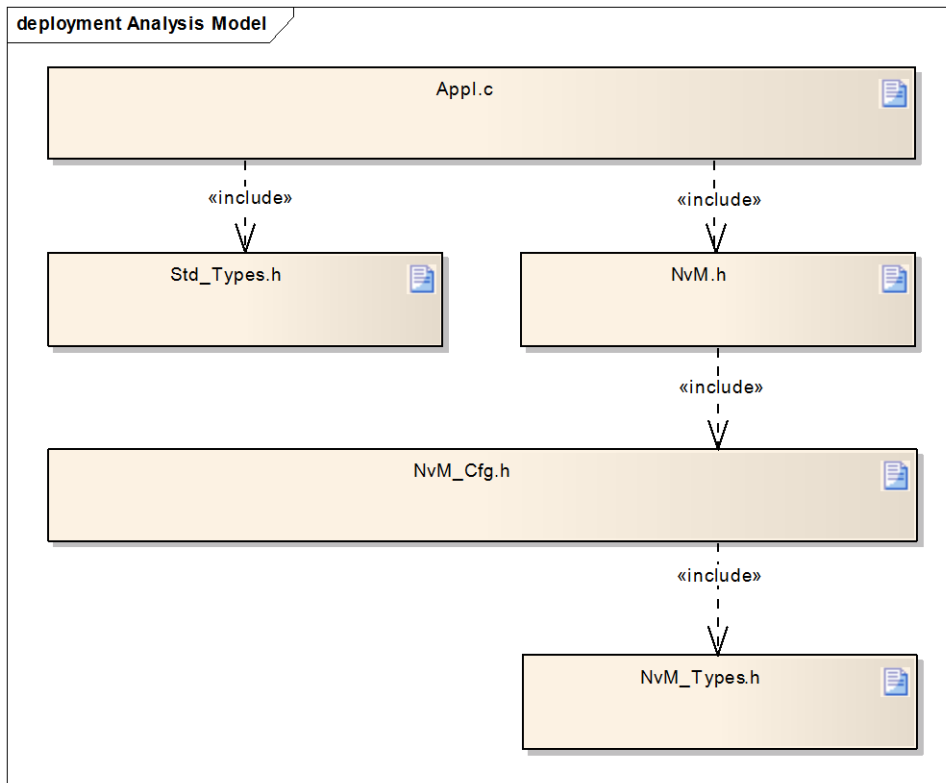


Figure 5-1 The file structure of the NVM sections module

5.4 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	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
Memory Mapping Sections											
NVM_START_SEC_CODE	■		■		■						
NVM_START_SEC_VAR_NOINIT_UNSPECIFIED			■								■
NVM_START_SEC_VAR_NOINIT_8			■								
NVM_START_SEC_VAR_UNSPECIFIED			■								
NVM_START_SEC_VAR_FAST_8				■							
NVM_START_SEC_CONST_UNSPECIFIED		■									
NVM_START_SEC_CONST_8		■									
NVM_START_SEC_CONST_16		■								■	
NVM_START_SEC_CONST_DESCRIPTOR_TABLE						■				■	
NVM_START_SEC_VAR_POWER_ON_INIT_UNSPECIFIED											■

Table 5-3 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.



Caution

The size of the section `NVM_START_SEC_CONST_DESCRIPTOR_TABLE` depends on the configuration settings. It makes sense to create an own section for this item if it becomes too big to link it into the same page/section as the elements of the MICROSAR NVM module. In this case the according memory modifier has to be used in order to address the elements in this section.

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. As these pragmas are already used when creating the NVM library (object code) these parameters are not link-time configurable. Libraries with different settings can be obtained at Vector Informatik GmbH. Please refer to the Software release notes (SRN) (or to the delivered `MemMap.h`, `Compiler.h/Compiler_Cfg.h`) for the settings made for your delivery.

`NVM_START_SEC_VAR_POWER_ON_INIT_UNSPECIFIED` shall be mapped to a section that is guaranteed to be zeroed out after Power-On Reset (therefore it may be a normal `ZERO_INIT` section, being zeroed out after any reset. Make sure this holds true for all

kinds of variable data (esp. uninitialized). If necessary, create a special section (don't map to a common one).

`NVM_START_SEC_VAR_UNSPECIFIED` shall also be mapped to a section that is guaranteed to be zeroed out. It holds the variable `NvM_TaskState_t` which has to be zero since NvM is not initialized.



Note

The integrator has to make sure specific section related settings (`#pragmas`) 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 (`#pragmas`) 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!

5.5 Dependencies on SW Modules

5.5.1 OSEK / AUTOSAR OS

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

5.5.2 DEM

NVM depends on an implementation of the DEM. It is used to report errors occurred during processing. The header file declaring the API must be configured via configuration tool.

5.5.3 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**.

5.5.4 MEMIF

The NVM uses configuration parameters defined by the MEMIF.

5.5.5 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.

5.5.6 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.

5.5.7 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`.

5.5.8 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.

5.6 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.

5.7 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.

5.7.1 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, 16bit pointers) to 12 bytes (32bit architectures, aligned structure members).

Additionally the setting **Internal Buffer for Crc Handling** 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 CRC, including CRC size. Sizes of NVRAM Blocks configured with **Use Synchronisation Mechanism**, will also be considered in calculation of internal buffer's size.

Additionally, each NVRAM block with **Calc RAM Block CRC** gets a dedicated RAM area for CRC storage, exactly matching CRC's size. As a result, applications' RAM blocks do not need to provide additional space for CRC. Therefore it does not affect RAM consumption.

5.7.2 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.

5.7.3 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 CRC 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

5.8 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.

5.8.1 NVM's provided Interfaces/Ports.

Every interface used by client SWCs needs to be remapped.

5.8.1.1 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.

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

For information about naming, please refer to 7.1.4

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.

5.8.2 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 7.1.5 for information the relationship between modelled callbacks (SWC’s P-Ports) and their RUNNABLES’ prototypes.

5.8.3 Request Result Types

In AUTOSAR4 new values for NvM_RequestResultType have been defined, namely NVM_REQ_REDUNDANCY_FAILED and NVM_REQ_RESTORED_FROM_ROM.

However, since their actual usage is not specified, they will not be used by NVM; its interface description omits them, and clients do not need to deal with them.

Finally, NvM uses the same set of request result values that was specified in AUTOSAR3. Therefore, this change in specification does not require any actions.

6 API Description

6.1 Interfaces Overview

For an interfaces overview please see Figure 3-2.

6.2 Type Definitions

Type Name	C-Type	Description	Value Range
NvM_RequestResult Type	uint8	An asynchronous API service can have following results or status that can be polled by <code>NvM_GetErrorStatus()</code> .	<p><code>NVM_REQ_OK</code> (see chapter 4.5.1)</p> <p>The last asynchronous request has been finished successfully. This is the default value after reset. This status has the value 0.</p> <p>Can be delivered by all asynchronous APIs.</p>
			<p><code>NVM_REQ_NOT_OK</code> (see chapter 4.5.1)</p> <p>The last asynchronous request has been finished unsuccessfully.</p> <p>Can be delivered by all asynchronous APIs.</p>
			<p><code>NVM_REQ_PENDING</code> (see chapter 4.5.1)</p> <p>An asynchronous request is currently being processed by the task.</p> <p>Can be delivered by all asynchronous APIs.</p>
			<p><code>NVM_REQ_INTEGRITY_FAILED</code> (see chapter 4.5.1)</p> <p>A NV block was supposed to be valid but it turned out that the data are corrupted (either CRC mismatch or the FEE or the EA reported an inconsistency).</p> <p>Can be delivered by <code>NvM_ReadBlock</code> or <code>NvM_ReadAll</code>.</p>
			<p><code>NVM_REQ_BLOCK_SKIPPED</code> (see chapter 4.5.1)</p> <p>The block was skipped during a multi block request.</p> <p>Can be delivered by <code>NvM_ReadAll</code> and <code>NvM_WriteAll</code>.</p>
			<p><code>NVM_REQ_NV_INVALIDATED</code> (see chapter 4.5.1)</p> <p>The NV block is marked as invalid.</p> <p>Can be delivered by <code>NvM_ReadBlock</code> or <code>NvM_ReadAll</code>.</p>
			<p><code>NVM_REQ_CANCELLED</code> (see chapter 4.5.1)</p>

Type Name	C-Type	Description	Value Range
			The last asynchronous <code>NvM_WriteAll()</code> has been cancelled by <code>NvM_CancelWriteAll()</code> .
<code>NvM_BlockIdType</code>	<code>uint16</code>	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 <code>NvM_GetErrorStatus()</code> and > Block ID 1 for the configuration Id block The block handles are created as defines in an ascending define list.	$\left\{ \begin{array}{l} [0..2^n[,]2^{15}..2^{15} + 2^n[,] \\ n = 16 - \text{NVM_DATASET_SELECTION_BITS} \end{array} \right\}$ <code>NVM_DATASET_SELECTION_BITS</code> 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 <code>NVM_DATASET_SELECTION_BITS</code> 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
<code>NvM_ServiceIdType</code>	<code>uint8</code>	Service Ids of the different service routines of the NVM.	<code>NVM_INIT (0u)</code> <code>NVM_SET_DATA_INDEX (1u)</code> <code>NVM_GET_DATA_INDEX (2u)</code> <code>NVM_SET_BLOCK_PROTECTION (3u)</code> <code>NVM_GET_ERROR_STATUS (4u)</code> <code>NVM_SET_RAM_BLOCK_STATUS (5u)</code> <code>NVM_READ_BLOCK (6u)</code> <code>NVM_WRITE_BLOCK (7u)</code> <code>NVM_RESTORE_BLOCK_DEFAULTS (8u)</code> <code>NVM_ERASE_BLOCK (9u)</code> <code>NVM_CANCEL_WRITE_ALL (10u)</code> <code>NVM_INVALIDATE_NV_BLOCK (11u)</code> <code>NVM_READ_ALL (12u)</code> <code>NVM_WRITE_ALL (13u)</code> <code>NVM_MAINFUNCTION (14u)</code> <code>NVM_GET_VERSION_INFO (15u)</code> <code>NVM_SET_BLOCK_LOCK_STATUS (16u)</code> The single values are applied as

Type Name	C-Type	Description	Value Range
			defines. See also chapter 4.5.1

Table 6-1 Type definitions

6.3 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

6.4 Services provided by NVM

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

6.4.1 NvM_Init

Prototype	
<code>void NvM_Init (void)</code>	
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 <code>NvM_ReadAll</code> request.	
Particularities and Limitations	
<ul style="list-style-type: none"> > This service is synchronous. > This service is non re-entrant. > This service is always available. 	
Expected Caller Context	
<ul style="list-style-type: none"> > 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 6-2 NvM_Init

6.4.2 NvM_SetDataIndex

Prototype
<code>Std_ReturnType NvM_SetDataIndex (NvM_BlockIdType BlockId, uint8 DataIndex)</code>


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	
<p>The request sets the specified index to associate a dataset NV block (with/without ROM blocks) with its corresponding RAM block. The <code>DataIndex</code> needs to have a valid value before a read/write/erase or invalidate request is initiated.</p> <p>If the dataset block has a set of ROM defaults, this function is used (prior to <code>NvM_ReadBlock()</code>) to select the appropriate ROM set.</p>	
Particularities and Limitations	
<ul style="list-style-type: none">> 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.	
<div>Note Usage of Explicit Synchronization, does not permit NvM's clients to issue new requests for a pending block.</div>	
Expected Caller Context	
<ul style="list-style-type: none">> This service is expected to be called in application context.	

Table 6-3 NvM_SetDataIndex

6.4.3 NvM_GetDataIndex

Prototype	
<pre>Std_ReturnType NvM_GetDataIndex (NvM_BlockIdType BlockId, uint8* DataIndexPtr)</pre>	
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	
<p>The request passes the current DataIndex (association) of the specified dataset block.</p>	

Particularities and Limitations
<ul style="list-style-type: none"> > 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
<ul style="list-style-type: none"> > This service is expected to be called in application context.

Table 6-4 NvM_GetDataIndex

6.4.4 NvM_SetBlockProtection


Prototype	
<pre>Std_ReturnType NvM_SetBlockProtection(NvM_BlockIdType BlockId, boolean ProtectionEnabled)</pre>	
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	
<ul style="list-style-type: none"> > 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. 	
<div>  <div> Note Usage of Explicit Synchronization, does not permit NvM's clients to issue new requests for a pending block. </div> </div>	
Expected Caller Context	
<ul style="list-style-type: none"> > This service is expected to be called in application context. 	

Table 6-5 NvM_SetBlockProtection

6.4.5 NvM_GetErrorStatus

Prototype	
<pre>Std_ReturnType NvM_GetErrorStatus (NvM_BlockIdType BlockId, NvM_RequestResultType* RequestResultPtr)</pre>	
Parameter	
BlockId	The Block identifier.
RequestResultPtr	Pointer where the result shall be written to. Result is of type <code>NvM_RequestResultType</code> . All possible results are described in chapter 6.2.
Return code	
E_OK	Request has been accepted.
E_NOT_OK	Request has not been accepted, e.g. Det error occurred.
Functional Description	
<p>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.</p> <p>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 <code>NvM_ReadAll()</code> and <code>NvM_WriteAll()</code> are multi block requests and change the status/error information of the multi block.</p>	
Particularities and Limitations	
<ul style="list-style-type: none">> 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	
<ul style="list-style-type: none">> This service is expected to be called in application context.	

Table 6-6 NvM_GetErrorStatus

6.4.6 NvM_GetVersionInfo

Prototype	
<pre>void NvM_GetVersionInfo (Std_VersionInfoType* versioninfo)</pre>	
Parameter	
versioninfo	Pointer to the address where the version info shall be written to.
Return code	
void	--
Functional Description	
<p>The request writes the version info (Vendor ID, module ID, SW major version, SW minor version, SW patch version) to the given pointer.</p>	
Particularities and Limitations	
<ul style="list-style-type: none">> 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 6-7 NvM_GetVersionInfo

6.4.7 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 6-8 NvM_SetRamBlockStatus

6.4.8 NvM_SetBlockLockStatus


Prototype	
<pre>void NvM_SetBlockLockStatus(NvM_BlockIdType BlockId, boolean BlockLocked)</pre>	
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	
<p>Service for setting/resetting the lock of a NV block.</p> <p>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.</p> <p>During processing of NvM_ReadAll, a locked NVRAM block shall be loaded from NV memory, regardless of RAM block's state (see 4.4.10). After that the lock is disabled again.</p> <p>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 4.4.17)</p>	
<div>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.</div>	
Particularities and Limitations	
<ul style="list-style-type: none">> 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	
<ul style="list-style-type: none">> This service is expected to be called by DCM.	

Table 6-9 NvM_SetBlockLockStatus

6.4.9 NvM_MainFunction

Prototype	
<code>void NvM_MainFunction (void)</code>	
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	
<ul style="list-style-type: none">> 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	
<ul style="list-style-type: none">> This service is expected to be called in application context.	

Table 6-10 NvM_MainFunction

6.4.10 NvM_ReadBlock

Prototype	
<code>Std_ReturnType NvM_ReadBlock (NvM_BlockIdType BlockId, void* NvM_DstPtr)</code>	
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 <code>NULL_PTR</code> 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 6-11 NvM_ReadBlock

6.4.11 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, <code>NULL_PTR</code> 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 a CRC, the RAM block CRC will be recalculated before the data and the CRC are written to the NV memory, even if the service `NvM_SetRamBlockStatus` was called before and the configuration was set that within this service, the CRC calculation should be done.

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 6-12 NvM_WriteBlock

6.4.12 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, <code>NULL_PTR</code> 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 6-13 NvM_RestoreBlockDefaults

6.4.13 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 6-14 NvM_EraseNvBlock

6.4.14 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 6-15 NvM_InvalidateNvBlock

6.4.15 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**: If disabled, all NVRAM blocks will be processed as described above, regardless of the result of reading/checking the configuration ID (match/mismatch/block invalid/integrity error/read failure). If **Dynamic Configuration Handling** is enabled, the NVM loads all NVRAM blocks as described above, only if it detected a configuration ID match. Otherwise (including failures) those blocks having option **Resistant to Changed Software** set will be loaded as if the configuration ID matched. The NVRAM blocks having this option cleared will be restored with ROM defaults, if available, and if **Select for ReadAll** was 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 6-16 NvM_ReadAll

6.4.16 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.



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 6-17 NvM_WriteAll

6.4.17 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 6-18 NvM_CancelWriteAll

6.4.18 NvM_KillWriteAll

Prototype

```
void NvM_KillWriteAll ( void )
```

Parameter

--	--
----	----

Return code	
void	--
Functional Description	
Request to cancel a running <code>NvM_WriteAll()</code> request destructively. To keep required wake-up response times in an ECU the ECUM has the possibility to time-out a non-destructive <code>NvM_CancelWriteAll()</code> request.	
Particularities and Limitations	
<ul style="list-style-type: none">> 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	
<ul style="list-style-type: none">> This service is expected to be called by ECUM	

Table 6-19 NvM_KillWriteAll

6.4.19 NvM_CancelJobs

Prototype	
<code>Std_ReturnType NvM_CancelJobs (NvM_BlockIdType BlockId)</code>	
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 pending job for a NV Block.	
Particularities and Limitations	
<ul style="list-style-type: none">> 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 <code>NVM_REQ_CANCELED</code>.	
Expected Caller Context	
<ul style="list-style-type: none">> This service is expected to be called in application context.	

Table 6-20 NvM_CancelJobs

6.4.20 NvM_RepairRedundantBlocks

Prototype
<code>void NvM_RepairRedundantBlocks (void)</code>

Parameter	
-	-
Return code	
-	-
Functional Description	
<p>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.</p> <p>If the NvM recognizes a lost redundancy, it will try to restore it via overwriting the defect block with data from valid block.</p> <p>Nothing to repair:</p> <ul style="list-style-type: none">- Both sub-blocks are readable- Both sub-blocks' Crcs match the recalculates Crc <p>Repairable blocks:</p> <ul style="list-style-type: none">- 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) <p>Non-repairable blocks:</p> <ul style="list-style-type: none">- Both sub-blocks aren't readable- Block sub-blocks' Crc does not match the recalculated Crc <p>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.</p>	
Particularities and Limitations	
<ul style="list-style-type: none">> 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	
<ul style="list-style-type: none">> This service is expected to be called in application context.	

Table 6-21 NvM_RepairRedundantBlocks

6.5 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
MEMIF	MemIf_GetStatus
MEMIF	MemIf_Cancel
MEMIF	MemIf_SetMode
CRC	Crc_CalculateCRC16
CRC	Crc_CalculateCRC32
EA	Used by MEMIF
FEE	Used by MEMIF

Table 6-22 Services used by the NVM

6.6 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.

6.6.1 NvM_JobEndNotification

Prototype	
<code>void NvM_JobEndNotification (void)</code>	
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	
<ul style="list-style-type: none"> > This service is synchronous. > This service is non re-entrant. 	
Expected Caller Context	
<ul style="list-style-type: none"> ■ The callback function <code>NvM_JobEndNotification</code> is intended to be used by the underlying memory abstraction (Fee/Ea) to signal end of job without error. 	

Table 6-23 NvM_JobEndNotification

6.6.2 NvM_JobErrorNotification

Prototype	
<code>void NvM_JobErrorNotification (void)</code>	
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	
<ul style="list-style-type: none"> > This service is synchronous. > This service is non re-entrant. 	
Expected Caller Context	
<ul style="list-style-type: none"> > The callback function <code>NvM_JobErrorNotification</code> is intended to be used by the underlying memory abstraction (Fee/Ea) to signal end of job with error. 	

Table 6-24 NvM_JobErrorNotification

6.7 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.

6.7.1 SingleBlockCallbackFunction

Prototype	
<pre>Std_ReturnType <SingleBlockCallbackFunction> (NvM_ServiceIdType ServiceId, NvM_RequestResultType JobResult)</pre>	
Parameter	
ServiceId	The service identifier (see chapter 6.2) 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 by the RTE or by a BSW component.

Call Context

- > Called from `NvM_MainFunction`
- > Asynchronous block processing completed (except `NvM_WriteAll`, for `NvM_ReadAll` it is configurable)

Table 6-25 SingleBlockCallbackFunction

6.7.2 MultiBlockCallbackFunction**Prototype**

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

Parameter

ServiceId	The service identifier (see chapter 6.2) of the completed request. <code>NvM_ServiceIdType</code> is of type <code>uint8</code> .
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` and `NvM_WriteAll`, respectively

Table 6-26 MultiBlockCallbackFunction

6.7.3 InitBlockCallbackFunction**Prototype**

```
Std_ReturnType <InitBlockCallbackFunction> ( void )
```



Parameter	
--	--
Return code	
Std_ReturnType	NVM always returns E_OK.
Functional Description	
Callback routine which shall be called by the NVM module to copy default data to a RAM block if a 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	
<ul style="list-style-type: none"> > This service is synchronous. > This service is non re-entrant 	
Call Context	
<ul style="list-style-type: none"> > Called from NvM_MainFunction > Called during processing of NvM_ReadAll, if application shall copy default values into the corresponding RAM block. 	

Table 6-27 InitBlockCallbackFunction

6.7.4 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	
<ul style="list-style-type: none"> > 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 6-28 Callback function for RAM to NvM copy

6.7.5 Callback function for NvM to RAM copy


Prototype	
<code>Std_ReturnType <NvM_ReadRamBlockFromNvm> (const void* NvMBuffer)</code>	
Parameter	
<code>NvMBuffer</code>	Internal RAM mirror where Ram block data can be read from
Return code	
<code>E_OK</code>	Callback function has been processed successfully
<code>E_NOT_OK</code>	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.	
<div>Note During calling <code>NvM_ReadRamBlockFromNvm</code> callback related block is still busy. No request for it shall be issued as long as block is busy.</div>	
Particularities and Limitations	
<ul style="list-style-type: none">> This service is synchronous.> This service is non re-entrant	
Call Context	
<ul style="list-style-type: none">> Called from <code>NvM_MainFunction</code>> Called during processing of NvM read requests, if application can copy data from internal RAM mirror to RAM block.	

Table 6-29 Callback function for NvM to RAM copy

6.8 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.

6.8.1 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 7.1.3 and 7.1.4.

6.8.1.1 Provide Ports on NVM side

At the Pports of the NVM the API functions described in 6.4 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:

6.8.1.1.1 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 6-30 Operations of Port Prototype Padmin_<BlockName>

6.8.1.1.2 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 6-31 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

6.8.1.2 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 6.7.

6.8.1.2.1 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 6-32 Operation of Port prototype NvM_RpNotifyFinished_Id<BlockName>

7 Configuration

7.1 Software Component Template

7.1.1 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.

7.1.2 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.

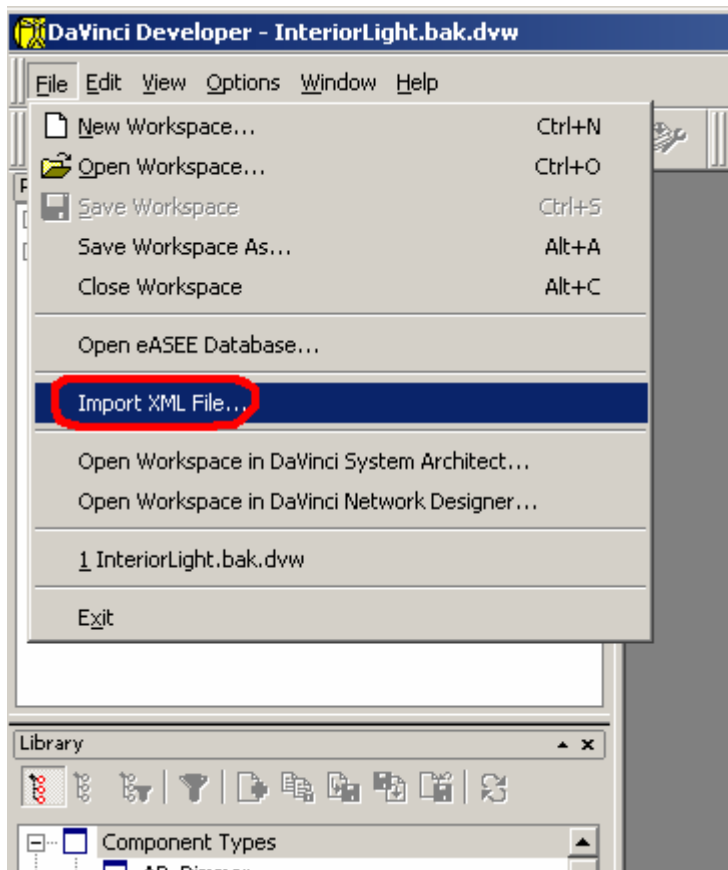


Figure 7-1 Import a new software component into DaVinci Developer

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.

7.1.3 Dependencies on Configuration of NVM Attributes

The configuration of the NVM attributes (described in chapter 7.1.5) 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.

7.1.3.1 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 6.8.1.1.
- > **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.

7.1.4 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.

7.1.4.1 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.

7.1.5 Modelling SWC's callback functions

According to AUTOSAR, the prototype of a SingleBlockCallbackFunction (Chapter 6.7.1), 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.

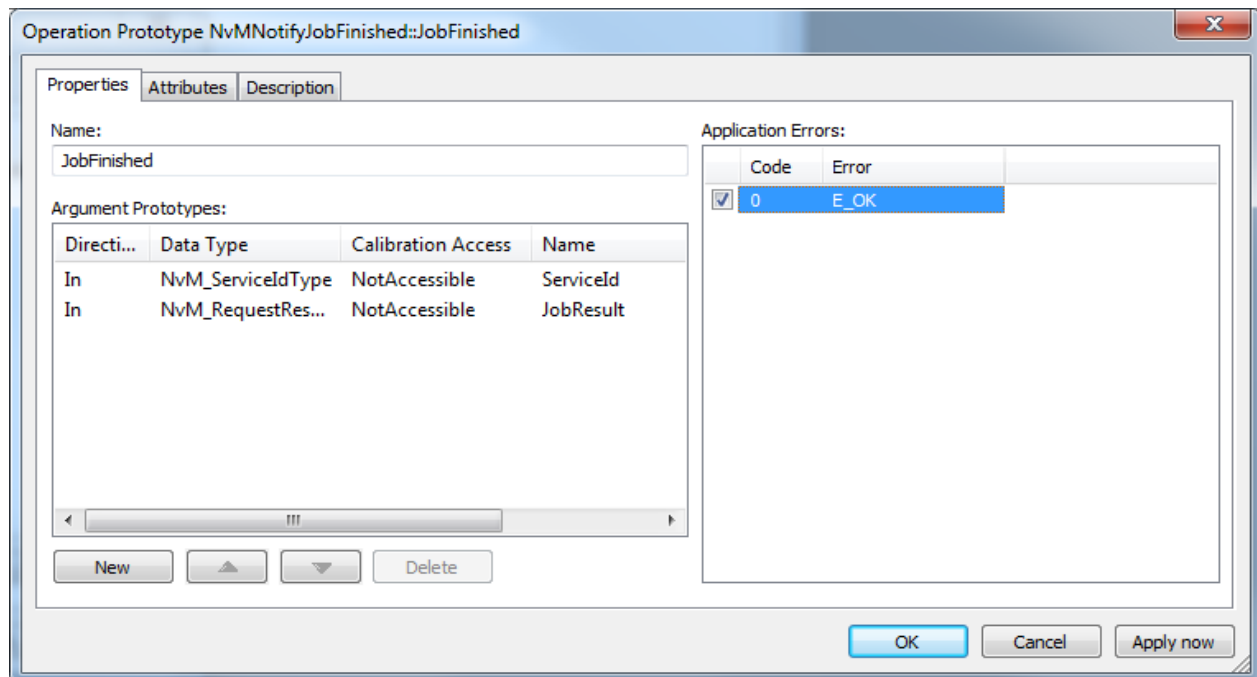


Figure 7-2 A "Single Block Job End Notification" with return type Std_ReturnType

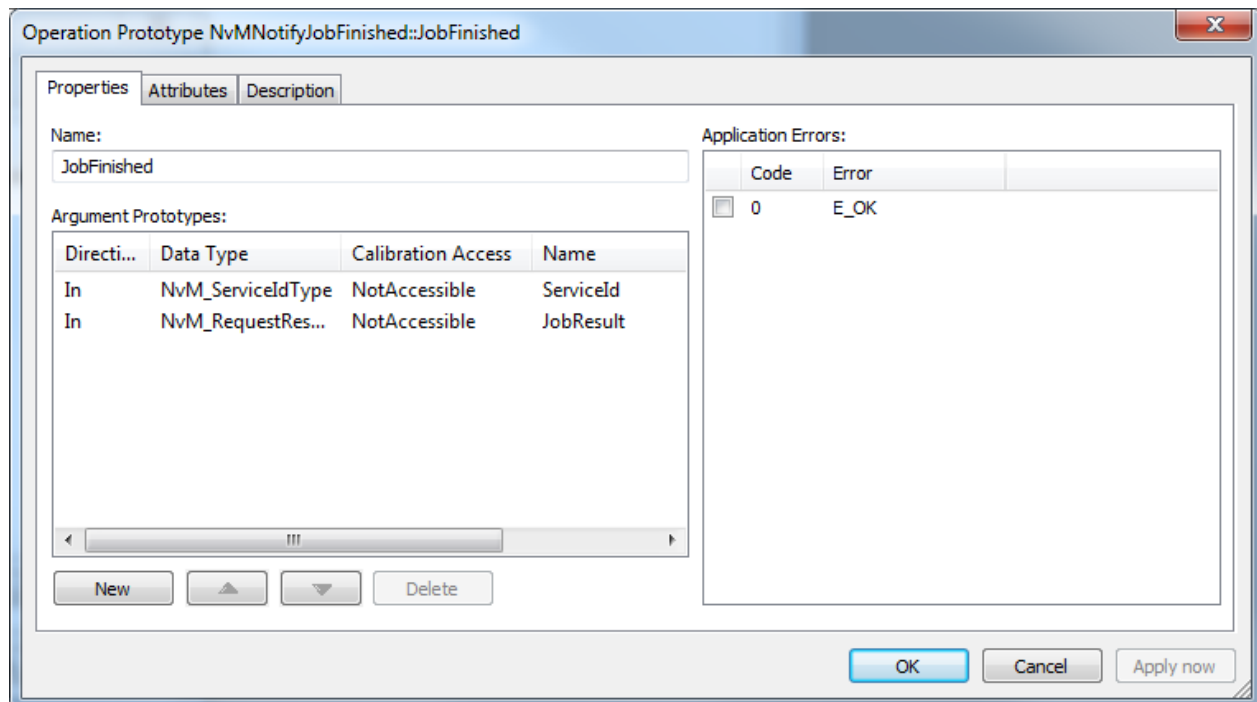


Figure 7-3 A "Single Block Job End Notification" with return type void.

NVM itself provides interfaces as described by Figure 7-3. 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.

7.2 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.

8 AUTOSAR Standard Compliance

8.1 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 4.4.18) cannot be limited by configuration. Rather those functions are expected to succeed within few attempts.

8.2 Additions/ Extensions

8.2.1 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 4.5.1.1.

8.2.2 Concurrent access to NV data

NVM provides for DCM possibility to access NV data concurrently with NVM application. (see chapter 4.4.17)

8.2.3 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 4.5.3.

8.2.4 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).

8.3 Limitations

There are no limitations.

9 Glossary and Abbreviations

9.1 Glossary

Term	Description
DaVinci Configurator Pro	Configuration and generation tool for MICROSAR.
DCM	Diagnostic Communication Manager
GCE	Generic Configuration Editor – generic tool for editing AUTOSAR configuration files. In DaVinci Configurator, the view can be switch to Generic Editor .
Per Instance Memory (PIM)	Memory (RAM) to be used by an instance of an Softwar Component, provide by RTE. It may also be used as permanent or tempary RAM block. Such a memory need is usually modeled using a tool like DaVinci Developer.
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 9-1 Glossary

9.2 Abbreviations

Abbreviation	Description
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BSW	Basis Software
CRC	Cyclic Redundancy Check
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
HIS	Hersteller Initiative Software
ISR	Interrupt Service Routine
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
PPort	Provide Port
RAM	Random Access Memory
ROM	Read Only Memory
RPort	Require Port
RTE	Runtime Environment
SRS	Software Requirement Specification
SWC	Software Component
SWS	Software Specification

Table 9-2 Abbreviations

10 Contact

Visit our website for more information on

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

www.vector.com