MICROSAR NVM

Technical Reference

Version 3.07.00

Authors	Manfred Duschinger
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 resons
Manfred Duschinger	2009-03-24	3.03.02	ESCAN00032480: Missing information in documentation: Chapter 6.4.5: 'Description of NvM_RequestResultType added'. Chapters 6.4.15 and 6.4.16: 'Services are multiblock requests'.
Manfred Duschinger	2009-06-03	3.04.00	ESCAN00032480: Update of History of version 3.03.02: Updated changed chapters. Chapter 6.2: 'Block ID 0 is only allowed for API NvM_GetErrorStatus()' ESCAN00033075: Chapter 4.5.1.1: DataIndex Check in NvM_ReadBlock() added. DataIndex Check was also added to NvM_InvalidateNvBlock() and NvM_EraseNvBlock(). ESCAN00033900: Chapter 4.4.17: "Priority Handling of DCM-Blocks" ESCAN00035089: Chapters 4.1, 7.2.2 "Callbacks NvM_JobEndNotification, NvM_JobErrorNotification implemented" ESCAN00034073: Chpaters 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

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

Illustrations

Tables

2 Component History

Component Version	New Features
3.05.xx	Calculated CRC32 value does not depend anymore on configuration of parameter NvmCrcNumOfBytes.
	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 NvM_SetBlockLockStatus.
	Storing each NVRAM block's CRC internally: RAM Blocks provided by the application don't have to allocate additional space for CRC.
	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*:	3	
Supported Configuration Variants:	link-time	
Vendor ID:	NVM_VENDOR_ID	30 decimal (= Vector-Informatik, according to HIS)
Module ID:	NVM_MODULE_ID	20 decimal (according to ref. [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.
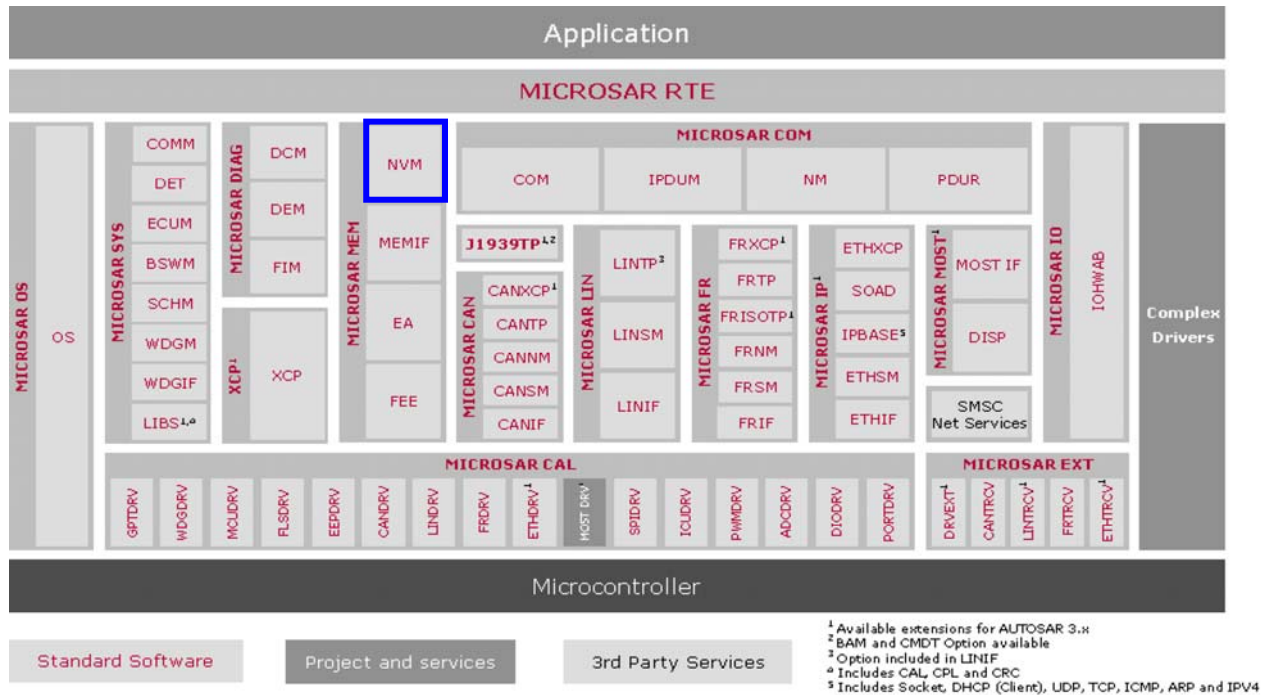
Figure 3-1 AUTOSAR architecture

The next figure shows the interfaces to adjacent modules of the NVM. These interfaces are described in chapter 6.
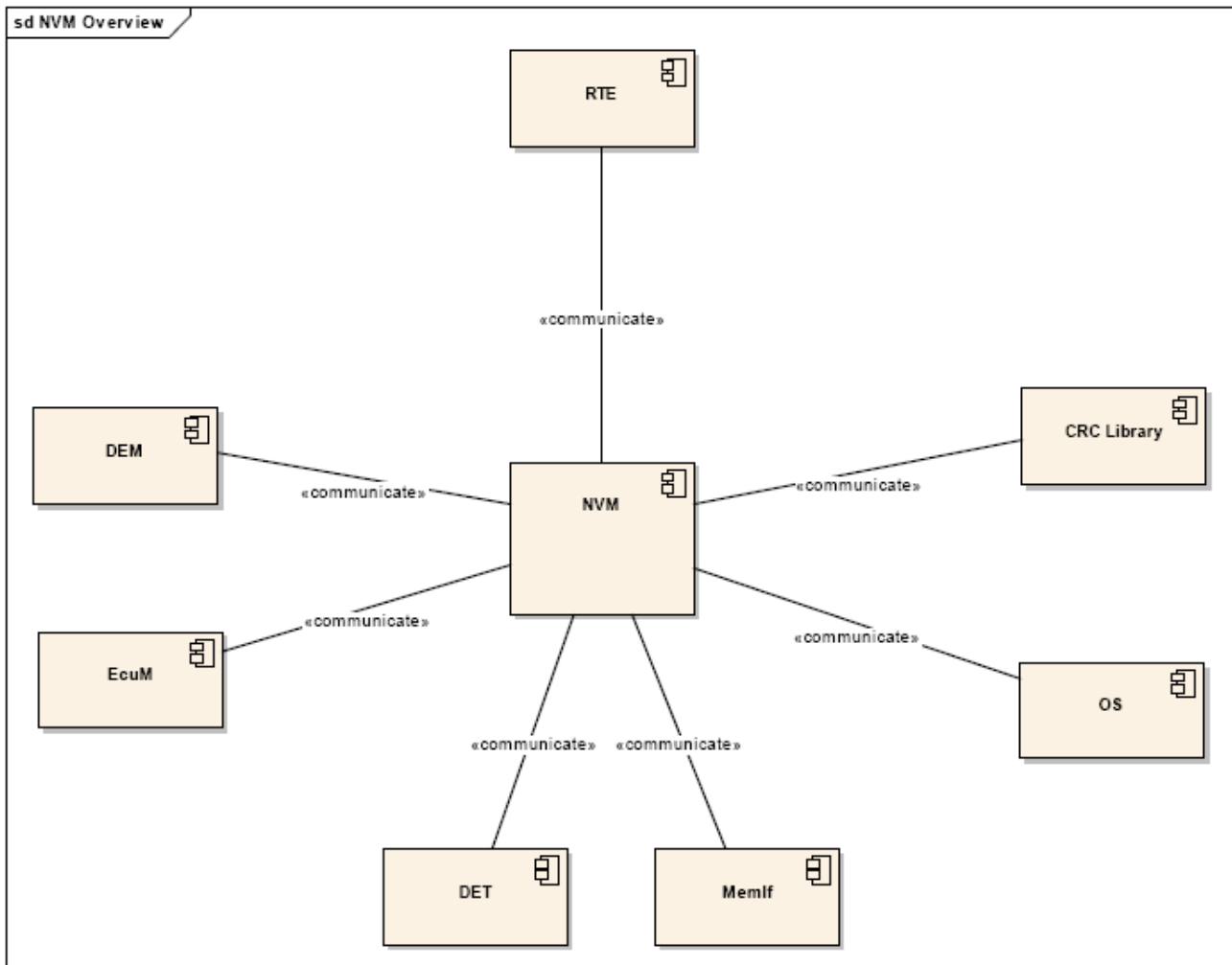
Figure 3-2 Interfaces to adjacent modules of the NVM

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

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

Table 4-2   Not supported SWS features

4.2 Initialization

Before the module NVM can be used it has to be initialized. All modules on which the NVM depends 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). The NVM "initialization" is twofold: `NvM_Init()` and `NvM_ReadAll()`. `NvM_ReadAll()` needs to be the last initialization step!

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

4.2.1 Block Size Checks

In Development Mode the NvM can be configured to check the size of every configured NVRAM block's permanent RAM block as well as its ROM block. It compares the configured NVRAM block length to the sizes of the RAM block and/or ROM block symbols.

If the check fails on one NVRAM block, an error is reported to the DET, and the NvM remains uninitialized.

> **Info**.
>
> If "Development Error Detection" is enabled, the RAM/ROM block length checks will always be generated into NvM_Cfg.c (`NvM_CfgCheckRomBlockLengths` and `NvM_CfgCheckRamBlockLengths`), and NvM_Init will always call them from NvM_Init.
>
> If RAM and/or ROM block checking was disabled, the corresponding function will be generated to always report "check passed" to the NvM. Hence these checks can be configured at link-time.
>
> Since the functions are "implemented" in generated NvM_Cfg.c, they can be debugged, even if NvM was delivered as library. Both checks always iterate over all configured blocks, i.e. they don't abort upon a detected mismatch. Accordingly the NVM always performs both checks, and then aborts in case of detected mismatch. This allows you to easily detect all mismatching RAM block lengths.

> **Caution**
>
> The NVM makes use of the `sizeof` operator. Therefore your ROM block and permanent RAM block symbols must be declared in the header files using complete types. Especially declarations like that are not allowed:
>
> - Arrays of unknown size (`[]`)
> - Structures of unknown contents (`struct x;`)

---

i | **Info**
By default the NvM performs strict checks, i.e. all sizes must exactly match. While too small RAM blocks usually will cause serious issues because the NVM would overwrite other SW's RAM, you would observe "loss of data" if they are too large, The NVM would read/write only parts of your data to NVM memory.

A common reason is given by CPU's alignment requirements: compilers may enlarge structures by adding padding bytes in order to align its structures on adequate boundaries.

4.2.2 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.

4.2.3 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 any more and have 'Select for ReadAll' (see chapter 7.2.4) set will be reloaded from NV memory or from ROM (if available).

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' (see chapter 7.2.2) 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.

Non-permanent RAM blocks, dataset blocks or blocks that are not selected for 'ReadAll' (configuration option) are skipped. They must be loaded manually by the application by calling `NvM_ReadBlock()`.

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()`

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()`

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`.

> **Info**
> The block handle names are automatically prefixed by the module short name followed by an underscore (NvM_). The prefixing has no influence on RTE.

The application only needs to provide space for CRC storage in its RAM block(s) when "Internal Buffer for Crc Handling" is disabled (see chapter 7.2.2). Is the internal buffer enabled, no additional space for CRC storage in the RAM block(s) is necessary. The NVM copies in this case the data not directly from the application variable to the NV memory. There is an extra buffering within the module NVM, because the NVM writes data and CRC, if configured, within one request to the NV memory. CRC is not stored in the application variable; it is stored in an internal variable by the NVM. Before processing write request to underlying components, the NVM copies data and CRC into the internal buffer.

> **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.

Native Block:

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

Redundant Block:

The data is stored twice in the NV area. 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.

If the NVM detects a defect 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.

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, 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 fix 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 init callback can be used. These features are selected during configuration. It is only possible to configure either ROM defaults or an init 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 is stored directly after the block data in the NV memory. In RAM, CRC is not stored after the block data; it is stored in an internal variable. NVM does not copy the data directly from the application variable to the NV memory. There is an extra buffering within the module NVM, because it writes data and CRC with one request to the NV memory. Thus an application does not need to provide any space for NvM's checksum at the end of a RAM block.

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 Asynchronous Job-End Notification

Alternatively to poll for the job-end, a job-end notification can be implemented and configured for every block. It will be called by the NVM every time a job is finished. Finished means: Job finished either successfully or cancelled.

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 prior 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 `NvM_SetRamBlockStatus(TRUE)` is called, the calculation of the CRC value over the RAM block's data is also initiated, unless the configuration option 'Calculate RAM CRC' (see chapter 7.2.4) was disabled for this block.

ⓘ	**Info** 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.
	Since CRC is quite time consuming, especially if performed frequently and/or over large data blocks, it might be more applicable to disable it using the option mentioned above. Then the data can only be re-used after a reset, if it did not change since last CRC recalculation, caused by a write request or after the last read request (if it was successful).

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.

> **Info**
> 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. For compatibility with older releases, this feature can be disabled for CRC32, which is not recommended, however.

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 can not 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 Callbacks

For any block ROM defaults or an initialization callback can be configured. The init callback is called every time the default values of the block are to be loaded, e.g. during a restore block defaults service.

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

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, this 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).

For accessing the alias of a NVRAM block, NVM provides the global macro `NvM_GetDcmBlockId()`. 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 is enabled. So cancellation of pending jobs by an immediate DCM-Block is avoided. The original priority itself is kept.

⚠	**Caution** DCM should lock the block with API NvM_SetBlockLockStatus (see chapter 6.4.8) before requesting the alias. In case of an error during job processing, DCM should also unlock the block again. In case of a successful job processing the block is automatically unlocked after next start-up (after NvM_ReadAll processing).
	Gets a block locked with NvM_SetBlockLockStatus, the original NVRAM block and the alias gets locked independent if the alias or the original block is requested.

4.4.18 Removed Functionality

Unlike in former versions of the NVM some functionality is no more the task of the NVM but of the FEE or the EA. Among this are:

- Walking block concepts

- Block allocation in memory

- Force erase boundaries (of NV devices)

- Methods for ensuring that corrupted data will not be signaled as valid

4.4.19 Changed Functionality

Unlike in former versions of the NVM some functionality is changed:

The checksum is stored directly after the block data only in the NV memory. In RAM, CRC is not stored after the block data; it is stored in an internal variable.

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`).

If another module is used for development error reporting, the function prototype for reporting the error can be configured by the integrator, but must have the same signature as the service `Det_ReportError()`.

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_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.
0x16	NVM_E_LIST_ OVERFLOW	All asynchronous requests can only be en-queued if the list is not full.
0x17	NVM_E_NV_WRITE_PROT ECTED	`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).
0x18	NVM_E_BLOCK_CONFIG	This service is not possible with this configuration.
0x0A	NVM_E_PARAM_BLOCK_ID	NVM API services may check, whether the passed `BlockId` is in the allowed range.
0x0B	NVM_E_PARAM_BLOCK_ TYPE	`NvM_SetDataIndex()` and `NvM_GetDataIndex()` are restricted to Dataset bocks. If these functions are called with any other bock type, this error code is produced. `NvM_RestoreBlockDefaults()` is restricted to blocks configured with ROM defaults or an init callback.
0x0C	NVM_E_PARAM_BLOCK_D ATA_IDX	`NvM_SetDataIndex()` may check the range of the passed `DataIndex`.
0x0D	NVM_E_PARAM_ADDRESS	A wrong pointer parameter was passed. (`NULL_PTR` passed in an asynchronous call, e.g. `NvM_WriteBlock()` for a non-permanent block)
0x0E	NVM_E_PARAM_DATA	A `NULL_PTR` was passed in one of the synchronous functions `NvM_GetDataIndex()`, `NvM_GetErrorStatus()` or `NvM_GetVersionInfo()`.
0x20	NVM_E_RAM_BLOCK_LEN GTH	At least one RAM block's size does not fit to the size that has been configured. May only be reported by `NvM_Init`.
0x21	NVM_E_ROM_BLOCK_LEN GTH	At least one ROM block's size does not fit to the size that has been configured. May only be reported by `NvM_Init`.

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. These checks are for development error reporting and can be en-/disabled separately. The configuration of en-/disabling the checks is described in chapter 7.2. En-/disabling of single checks is an

addition to the AUTOSAR standard which requires to en-/disable the complete parameter checking via the parameter NVM_DEV_ERROR_DETECT.

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

Service \ Check	Module's initialization	Block's Management Type check	Block's Write Protection check	Block's Pending State check	Block Id check	DataIndex check	Pointers check	Block length 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()	■							

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. Please refer to chapter 7.2.5.

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

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

Table 4-7   Errors reported to DEM

Both 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.

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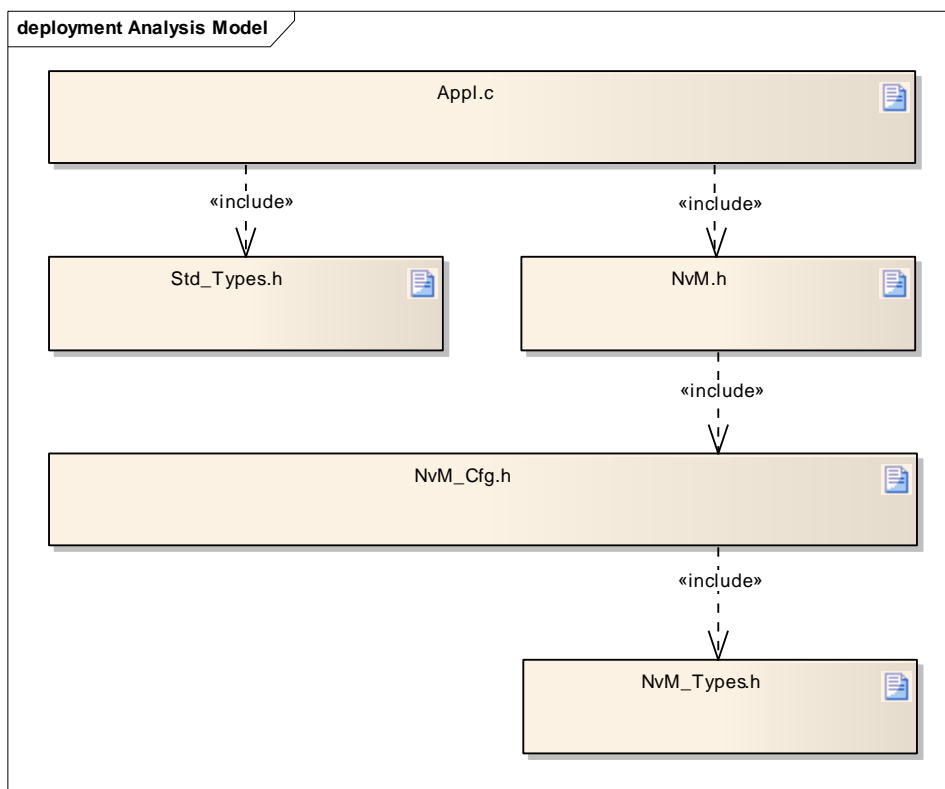
Figure 5-1 The file structure of the NVM sections module

5.3 Compiler Abstraction and Memory Mapping

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

The following table contains the memory section names and the compiler abstraction definitions defined for the NVM and illustrates their assignment among each other.

Compiler Abstraction Definitions / Memory Mapping Sections	NVM_PRIVATE_CODE	NVM_PRIVATE_CONST	NVM_PRIVATE_DATA	NVM_FAST_DATA	NVM_PUBLIC_CODE	NVM_PUBLIC_CONST	NVM_APPL_CODE	NVM_APPL_CONST	NVM_APPL_DATA	NVM_CONFIG_CONST	NVM_CONFIG_DATA
NVM_START_SEC_CODE	■				■						
NVM_START_SEC_ VAR_NOINIT_UNSPECIFIED			■								■
NVM_START_SEC_VAR_NOINIT_8 BIT			■								
NVM_START_SEC_VAR_UNSPECIF IED			■								
NVM_START_SEC_VAR_FAST_8BI T				■							
NVM_START_SEC_ CONST_UNSPECIFIED		■				■					
NVM_START_SEC_CONST_8BIT		■									
NVM_START_SEC_CONST_16BIT										■	
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.

> **Caution**
> The sections configured above have to fit to the link file configuration as well as to the memory modifier settings in the Compiler Abstraction!

5.4 Dependencies on SW Modules

5.4.1 OSEK / AUTOSAR OS

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

5.4.2 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.

5.4.3 DET

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

5.4.4 MEMIF

The NVM uses configuration parameters defined by the MemIf.

5.4.5 CRC Library

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

> **Info**
> Since the "Configuration Id Block" (see also chapter 7.2.3) must be configured with either CRC16 or CRC32; you will always need the CRC library.

5.4.6 Callback Functions

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

callback function declarations must be made within the application and be included by the NVM.

5.4.7 RTE

When at least one Service Port is enabled (see chapter 7.2.4) and corresponding PIM (see TechnicalReference of RTE) is available, all additional necessary header files are included automatically. SWC must not include `NvM.h`.

5.5 Integration Steps

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

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

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

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

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

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

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

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

- Make sure that the initialization sequence is correct. FEE/EA and FLS/EEP must be initialized before any NVM request (usually `NvM_ReadAll()`) can be used.

- 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.

5.6 Estimating Resource Consumption

Besides resources needed anyway when using NVM, there are some configuration options influencing resource consumption of your system. In general these options affect usage independently of the number of configured NVRAM blocks. Additionally each NVRAM

block itself requires resources in RAM, ROM and NV, respectively. The following sections will summarize the options and give you hints, how to estimate their effects.

5.6.1 RAM Usage

In general, each NVRAM block consumes RAM – for the application-defined RAM-block as well as for the internal block management structure, which holds information about request results, blocks' attributes and the data indexes. The amount of RAM occupied by the RAM block itself should equal 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.

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 equals the size of largest configured NVRAM block with CRC, having a permanent RAM block associated, including CRC size.

Additionally, each NVRAM block with CRC automatically 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.

5.6.2 ROM Usage

The largest amount of ROM resources being needed depends on the number (and sizes) of blocks configured with ROM defaults. The remarks about RAM blocks also apply to ROM blocks. Especially for block management type 'DATASET_ROM'. The size of the object (variable) declared by application must be considered, rather than the configured NVRAM block length.

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. It can be from 18 bytes (16bit architecture, 16bit pointers) to 44 bytes (32bit architecture, 32bit pointers, structure members aligned).

There are some configuration options affecting NVM's 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.

5.6.3 NV Usage

The requirements on NV memory space per device are affected by the NVRAM blocks and their configuration. Basically, each NV block allocates as many bytes as specified for its length, plus 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 ore more blocks consuming NV space:

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

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 `NvM_GetErrorStatus()`.	`NVM_REQ_OK` (see chapter 4.5.1) The last asynchronous request has been finished successfully. This is the default value after reset. This status has the value 0.
			`NVM_REQ_NOT_OK` (see chapter 4.5.1) The last asynchronous request has been finished unsuccessfully.
			`NVM_REQ_PENDING` (see chapter 4.5.1) An asynchronous request is currently being processed by the task.
			`NVM_REQ_INTEGRITY_FAILED` (see chapter 4.5.1) 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).
			`NVM_REQ_BLOCK_SKIPPED` (see chapter 4.5.1) The block was skipped during a multi block request.
			`NVM_REQ_NV_INVALIDATED` (see chapter 4.5.1) The NV block is marked as invalid.
			`NVM_REQ_CANCELLED` (see chapter 4.5.1) The last asynchronous `NvM_WriteAll()` has been cancelled by `NvM_CancelWriteAll()`.

Type Name	C-Type	Description	Value Range

NvM_BlockIdType	uint16	It is the type of a block handle that is used by the application in order to access a NVM block. There are two reserved IDs:
		
		■ Block ID 0 for multi block requests (Block ID 0 is only allowed for API NvM_GetErrorStatus()) and
		
		■ Block ID 1 for the configuration Id block
		
		The block handles are created as defines in an ascending define list.
			`[0`
			`2^(16-`
			`NVM_DATASET_SELECTION_BITS)-`
			`1]`
			
			`NVM_DATASET_SELECTION_BITS` is the maximum number of bits that are needed in order to store the maximum dataset value.
			
			Example:
			
			The dataset block with the greatest number of datasets has six of them. So it is necessary to store the data index 0…5 to select the appropriate dataset block. To store the value five, three bits are necessary. So `NVM_DATASET_SELECTION_BITS` has the value 3.
			
			This means that only the block IDs 0…2^(16-3)-1 = 2…8191 are available as block handles.

NvM_ServiceIdType	uint8	Service Ids of the different service routines of the NVM.	`NVM_INIT (0u)`
			`NVM_SET_DATA_INDEX (1u)`
			`NVM_GET_DATA_INDEX (2u)`
			`NVM_SET_BLOCK_PROTECTION (3u)`
			`NVM_GET_ERROR_STATUS (4u)`
			`NVM_SET_RAM_BLOCK_STATUS (5u)`
			`NVM_READ_BLOCK (6u)`
			`NVM_WRITE_BLOCK (7u)`
			`NVM_RESTORE_BLOCK_DEFAULTS (8u)`
			`NVM_ERASE_BLOCK (9u)`
			`NVM_CANCEL_WRITE_ALL (10u)`
			`NVM_INVALIDATE_NV_BLOCK (11u)`
			`NVM_READ_ALL (12u)`
			`NVM_WRITE_ALL (13u)`
			`NVM_MAINFUNCTION (14u)`
			`NVM_GET_VERSION_INFO (15u)`
			`NVM_SET_BLOCK_LOCK_STATUS (16u)`
			
			The single values are applied as 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	
void **NvM_Init** (void)	
Parameter	
--	--
Return code	
void	--
Functional Description	
Service for basic NVM initialization. (Exclusively called by ECU State Manager). The time consuming NVRAM block initialization and setup according to the block descriptor is done by the `NvM_ReadAll` request.	
Particularities and Limitations	
■ This service is synchronous. ■ This service is non re-entrant. ■ This service is always available.	
Expected Caller Context	
■ This service is expected to be called in application context.	

Table 6-2   NvM_Init

6.4.2 NvM_SetDataIndex

Prototype	
void **NvM_SetDataIndex** (NvM_BlockIdType BlockId, uint8 DataIndex)	
Parameter	
BlockId	The Block identifier.
DataIndex	Index position of a Block in the NV Block of Dataset type.
Return code	
void	--
Functional Description	
The request sets the specified index to associate a dataset NV block (with/without ROM blocks) with its corresponding RAM block. The DataIndex needs to have a valid value before a read/write/erase or invalidate request is initiated. If the dataset block has a set of ROM defaults, this function is used (prior to NvM_ReadBlock()) to select the appropriate ROM set.	
Particularities and Limitations	
■ This service is synchronous. ■ This service is re-entrant. ■ This service is available if API configuration class 2 or 3 is configured. ■ The NVRAM manager shall have been initialized before this request is called.	
Expected Caller Context	
■ This service is expected to be called in application context.	

Table 6-3   NvM_SetDataIndex

6.4.3 NvM_GetDataIndex

Prototype	
void **NvM_GetDataIndex** (NvM_BlockIdType BlockId, uint8* DataIndexPtr)	
Parameter	
BlockId	The Block identifier.
DataIndexPtr	Address where the current DataIndex shall be written to
Return code	
Void	--
Functional Description	
The request passes the current DataIndex (association) of the specified dataset block.	
Particularities and Limitations	
■ This service is synchronous. ■ This service is re-entrant. ■ This service is available if API configuration class 2 or 3 is configured. ■ The NVRAM manager shall have been initialized before this request is called.	

Expected Caller Context
■ This service is expected to be called in application context.

Table 6-4   NvM_GetDataIndex

6.4.4   **NvM_SetBlockProtection**

Prototype	
void **NvM_SetBlockProtection** (NvM_BlockIdType BlockId, boolean ProtectionEnabled)	
Parameter	
BlockId	The Block identifier.
ProtectionEnabled	This parameter is responsible for setting the write protection of a selected NVRAM block: TRUE: enable protection FALSE: disable protection
Return code	
void	--
Functional Description	
The request sets the write protection for the NV block. Any further write/erase/invalidate requests to the NVRAM block are rejected synchronously if the NV block-write protection is set. The data area of the RAM block remains writable in any case.	
Particularities and Limitations	
■ This service is synchronous. ■ This service is re-entrant. ■ This service is available if API configuration class 3 is configured. ■ The NVRAM Manager shall have been initialized before this request is called. The protection can not be released for a write once block that has already been written.	
Expected Caller Context	
■ This service is expected to be called in application context.	

Table 6-5   NvM_SetBlockProtection

6.4.5 NvM_GetErrorStatus

Prototype	
void **NvM_GetErrorStatus** (NvM_BlockIdType BlockId, uint8* RequestResultPtr)	
Parameter	
BlockId	The Block identifier.
RequestResultPtr	Pointer where the result shall be written to.
	Result is of type NvM_RequestResultType. All possible results are described in chapter 6.2.
Return code	
void	--
Functional Description	
The request reads the block dependent error/status information and writes it to the given address. The status/error information was set by a former or current asynchronous request. This API can also be requested with BlockId 0 (multi block). Then the multi block error/status information will be read to the given address. Only NvM_ReadAll() and NvM_WriteAll() are multi block requests and change the status/error information of the multi block.	
Particularities and Limitations	
■ This service is synchronous. ■ This service is re-entrant. ■ This service is always available. ■ The NVRAM Manager shall have been initialized before this request is called.	
Expected Caller Context	
■ This service is expected to be called in application context.	

Table 6-6   NvM_GetErrorStatus

6.4.6 NvM_GetVersionInfo

Prototype	
void **NvM_GetVersionInfo** (Std_VersionInfoType* versioninfo)	
Parameter	
versioninfo	Pointer to the address where the version info shall be written to.
Return code	
void	--
Functional Description	
The request writes the version info (Vendor ID, module ID, Instance ID, SW major version, SW minor version, SW patch version) to the given pointer.	
Particularities and Limitations	
■ This service is synchronous. ■ This service is re-entrant. ■ This service is available if the pre-compile switch "Use version info API" is enabled.	

Table 6-7   NvM_GetVersionInfo

6.4.7   NvM_SetRamBlockStatus

Prototype
void **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	
Void	--

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.

Info
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

based on template version 3.01

6.4.8   NvM_SetBlockLockStatus

Prototype	
Std_ReturnType **NvM_SetBlockLockStatus** (NvM_BlockIdType BlockId, boolean Locked)	
Parameter	
BlockId	The Block identifier.
Locked	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	
E_OK	Request has been accepted.
E_NOT_OK	Request has not been accepted.
Functional Description	
Service for setting/resetting the lock of a NV block. The NV contents associated to the NVRAM block identified by BlockId, will not be modified by any request. The Block is skipped during NvM_WriteAll. Other requests, that are NvM_WriteBlock, NvM_InvalidateNvBlock, NvM_EraseNvBlock, return without error notification to Det or Dem. During processing of NvM_ReadAll, this NVRAM block shall be loaded from NV memory. After loading the block from NV memory the lock is disabled again. If a block gets locked with NvM_SetBlockLockStatus, the original NVRAM block and the alias is locked. The lock is independent on the requested BlockId, original one or DCM BlockId. (see chapter 4.4.17)	

> **Info**
> This function is an addition to the standard AUTOSAR specification.

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

Table 6-9   NvM_SetBlockLockStatus

6.4.9   NvM_MainFunction

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

Table 6-10 NvM_MainFunction

6.4.10   NvM_ReadBlock

Prototype	
Std_ReturnType **NvM_ReadBlock** (NvM_BlockIdType BlockId, uint8* NvM_DstPtr)	
Parameter	
BlockId	The Block identifier.
NvM_DstPtr	Pointer where the data of a non-permanent RAM block shall be written to. If the block is permanent NULL_PTR shall be passed.
Return code	
E_OK	Request has been accepted.
E_NOT_OK	Request has not been accepted, e.g. because of a list overflow.
Functional Description	
Request to copy the data of the NV block to its corresponding RAM block. This function queues the read request and returns the acceptance result synchronously. The NVM can notify the application by callback when the service is finished.	
Particularities and Limitations	
■ This service is asynchronous.	
■ This service is re-entrant.	
■ This service is available if API configuration class 2 or 3 is configured.	
■ The NVRAM Manager shall have been initialized before this request is called. In development mode the service will not accept the call if the block is already queued (either for this or for a different service).	

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 uint8* NvM_SrcPtr)

Parameter	
BlockId	The Block identifier.
NvM_SrcPtr	Pointer where the data of a non-permanent RAM block shall be read from. If the block is permanent, NULL_PTR shall be passed.

Return code	
E_OK	Request has been accepted.
E_NOT_OK	Request has not been accepted, e.g. list overflow.

Functional Description
Request for copying data from the RAM block to its corresponding NV block. This function queues the write request and returns the acceptance result synchronously.
If the block has 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.

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,` `                                    uint8* NvM_DstPtr )`

Parameter	
`BlockId`	The Block identifier.
`NvM_DstPtr`	Pointer where the data of a non-permanent RAM block shall be written to. If the block is permanent, `NULL_PTR` shall be passed.

Return code	
`E_OK`	Request has been accepted.
`E_NOT_OK`	Request has not been accepted, e.g. list overflow.

Functional Description
Request to copy the ROM block default data to its corresponding RAM block. The selected block needs either ROM defaults or an initialization callback.
This function queues the restore request and returns the acceptance result synchronously.
The NVM can notify the application by callback when the service is finished.

Particularities and Limitations
■ This service is asynchronous.
■ This service is re-entrant.
■ This service is available if API configuration class 2 or 3 is configured.
■ The NVRAM Manager shall have been initialized before this request is called. In development mode the service will not accept the call if the block is already queued (either for this or for a different service). This function is not intended for reading ROM sets of a dataset ROM block. Use `NvM_ReadBlock()` instead for these blocks.

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!

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.	
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" (see chapter 7.2.2): 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" (see chapter 7.2.4) 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.

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 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).

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.

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 `NvM_WriteAll()` request destructively. To keep required wake-up response times in an ECU the ECUM has the possibility to time-out a non-destructive `NvM_CancelWriteAll()` request.

Particularities and Limitations
■ This service is synchronous.
■ This service is non re-entrant.
■ This service is available if the pre-compile switch "NvmKillWriteAllApi" (only in Generic Editor in container "Nvm_30_CommonVendorParams") is enabled independent on API configuration class.
■ The NVRAM Manager shall have been initialized before this request is called.

Expected Caller Context
■ This service is expected to be called by ECUM

Table 6-19 NvM_KilllWriteAll

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 6-20 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	
void **NvM_JobEndNotification** (void)	
Parameter	
-	-
Return code	
void	-
Functional Description	
Function to be used by the underlying memory abstraction to signal end of job without error.	
Particularities and Limitations	
■ This service is synchronous. ■ This service is non re-entrant.	
Expected Caller Context	
■ The callback function NvM_JobEndNotification is intended to be used by the underlying memory abstraction (Fee/Ea) to signal end of job without error.	

Table 6-21 NvM_JobEndNotification

6.6.2   NvM_JobErrorNotification

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

Table 6-22 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 appropriate function prototype signatures, which are described in the following sub-chapters.

6.7.1   SingleBlockCallbackFunction

Prototype	
Std_ReturnType **<SingleBlockCallbackFunction>** (NvM_ServiceIdType ServiceId, NvM_RequestResultType JobResult)	
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.	

Call Context
■ Called from `NvM_MainFunction`
■ Asynchronous block processing completed (except `NvM_WriteAll`, for `NvM_ReadAll` it is configurable)

Table 6-23 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. `NvM_ServiceIdType` is of type `uint8`.
JobResult	Result of the multi block job.
Return code	
void	--
Functional Description	
Common callback routine to notify the upper layer that an asynchronous multi block request has been finished.	
Particularities and Limitations	
■ This service is synchronous.	
■ This service is non re-entrant.	
Call Context	
■ Called from `NvM_MainFunction`.	
■ Called upon completion of `NvM_ReadAll` and `NvM_WriteAll`, respectively	

Table 6-24 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.	
Particularities and Limitations	
■ This service is synchronous.	
■ This service is non re-entrant	

Call Context
■ Called from `NvM_MainFunction`
■ Called during processing of `NvM_ReadAll`, if application shall copy default values into the corresponding RAM block.

Table 6-25 InitBlockCallbackFunction

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 6-26 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 6-27 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 6-28 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 can be generated on demand by DaVinci Configurator via the menu "Tools→AUTOSAR→SW-C→NvM". Additionally, DaVinci Configurator Pro also generates this file at the end of each generation process automatically. The target directory for SW-C files can be set via the menu "Tools→Preferences→NvM→SW-C Files".
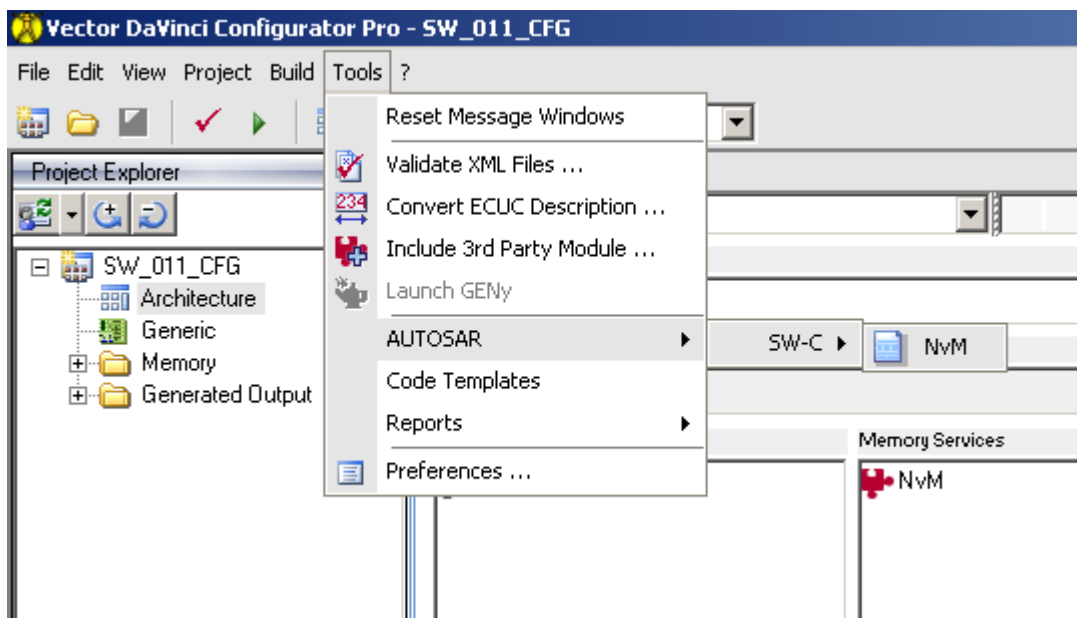

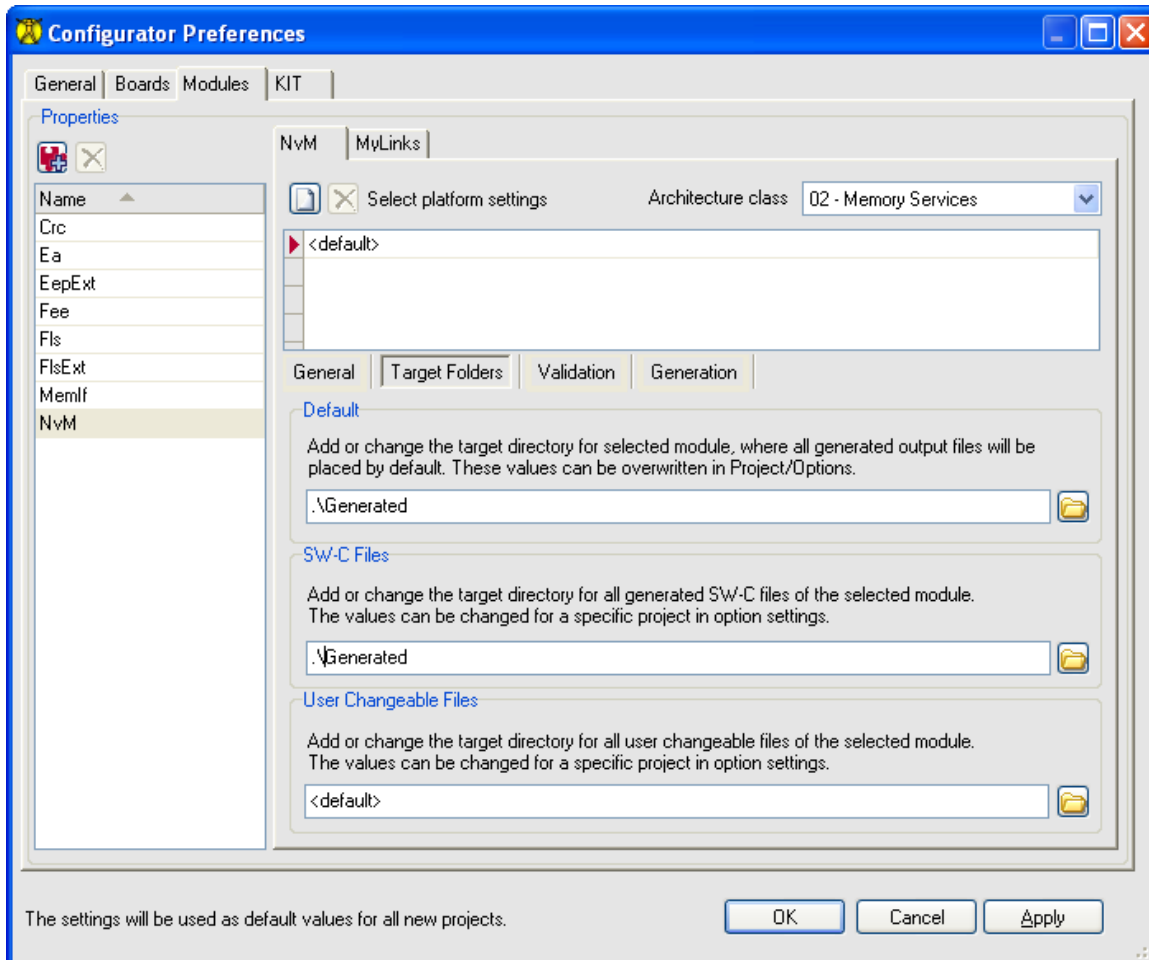Figure 7-1 Generate an NVM software component template

Figure 7-2 Change target directory for all generated SW-C files of NvM.

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.
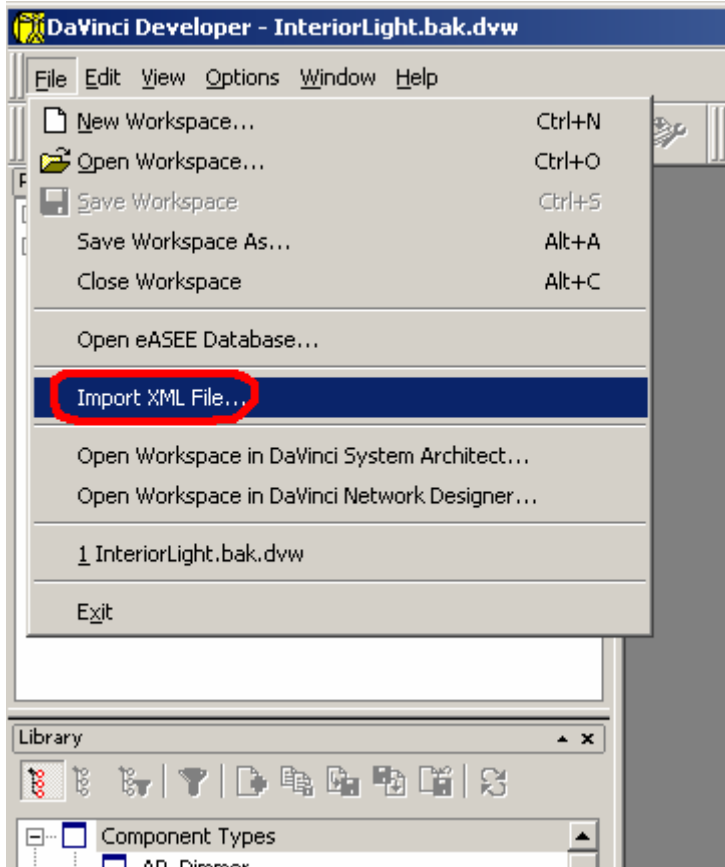

Figure 7-3 Import a new software component into DaVinci Developer

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 7.2) 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 'Srv'.

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 'NvM_Administration' (only in 'API Configuration Class 3')

- 'PS' → Linked with Port Interface 'NvM_AC{1|2|3}[_SRBS][_Defs][_DS]_Srv'. The actual interface depends on the possibilities described above.

- 'NvM_RpNotifyFinished' → Linked with Port Interface NvM_NotifyJobFinished 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.

7.2   Configuration of NVM Attributes

In the NVM the attributes can be configured with the following methods:

- Configuration in DaVinci Configurator

> ⚠️ **Caution**
> Because AUTOSAR forbids the use of the `sizeof`-operator in production code, 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.

7.2.1 Start configuration of the NVM

The component name of the NVRAM Manager in DaVinci Configurator is "NvM". In the "Architecture view" (initial page) of DaVinci Configurator the NvM can be opened by its context menu to start its configuration. The NvM can also be opened for configuration using the Project Explorer, selecting Memory → Services → NvM.

7.2.2 General Settings

Attribute Name	Configuration Variant	Value Type	Values The default value is written in bold	Description
SWC Settings				
Service Component Name	link-time	string	Valid C-name, maximum length: 20 characters, default: **NvM**	Create a unique name space for the NVM. This name is used for SWC Description generation. The generated component will get this name. Several names will be prefixed with this name.
Long Service Port Names	link-time	bool	**ON** OFF	Enables long service port names; Each port prototype's name is post fixed by '{BlockName}' instead of '{BlockId}'
Queues and API configuration				

API Configuration Class	pre-compile	--	Class 1
			Class 2
			Class 3
				Different classes with API function sets can be selected.
				By setting this preprocessor switch to 'Class 1' the following options are disabled:
				`NVM_JOB_PRIORISATION`
				`NVM_SIZE_STANDARD_JOB_QUEUE`
				`NVM_SIZE_IMMEDIATE_JOB_QUEUE`

Use Version Info API	pre-compile	--	ON **OFF**	Use `NvM_GetVersionInfo()`.
Use Set Ram Block Status API	pre-compile	--	**ON** OFF	Use `NvM_SetRamBlockStatus()`.
Job Prioritization	pre-compile	--	ON **OFF**	Enable the prioritized queuing of jobs.
Size of Standard Job Queue	link-time	integer	1.. **8** ..254	Size of the standard job queue.
Size of Immediate Job Queue	link-time	integer	**1** ..254	Size of the immediate job queue.
Configuration option				
ID of this Configuration Set	link-time	integer	0.. **1** ..65535	Compiled configuration set identifier.
Dynamic Configuration Handling	pre-compile	--	ON **OFF**	Enable/disable dynamic configuration handling.

Attribute Name	Configuration Variant	Value Type	Values The default value is written in bold	Description
Further settings				
Drive Mode	pre-compile	--	ON **OFF**	If this option is selected, the drive mode of the underlying memory devices will be set to fast during multi block services.
Polling Mode	pre-compile	--	**ON** OFF	If this option is selected, NVM polls the underlying layer till job completion. The callback functions are not used by the lower layer and must be disabled in this layer.
Internal Buffer for Crc Handling	pre-compile	--	**ON** OFF	Enable and disable internal Crc Buffer. If Internal Crc Buffer is enabled, NvM handles Crc in an internal buffer. Memory for Crc must not be allocated at the end of Ram Block. If internal Crc Buffer is disabled, Memory for Crc at the end of Ram Block is necessary!
CRC Bytes per Cycle	link-time	integer	1.. **64** .. 65535	Maximum number of bytes of a CRC checksum that are calculated during one cycle of `NvM_MainFunction`.
Max. Number of Write Retries	link-time	integer	0.. **1** ..7	Number of retries until a write job fails.
Dataset Selection Bits	link-time	integer	1.. **4** ..8	Defines the number of least significant bits which are used to address a dataset of a block of type dataset (see chapter 4.4.5.2).
Critical Sections				
Function Set	link-time	--	all named sets defined in 'MyECU', default: **UseSuspendFunctions**	The different services to protect critical sections can be defined in 'MyECU'.

Table 7-1   General Settings

7.2.3 Special NVRAM Blocks

Attribute Name	Configuration Variant	Value Type	Values The default value is written in bold	Description
Config ID Block				
Name	link-time	C-identifier	Valid C-identifier, default: **ConfigBlock**	Logical name of the reserved configuration block to be used within the application.
Device	link-time	FEE/EA or third party module	All configured NV abstraction devices (FEE/EA …), no default.	The device (Memory Hardware Abstraction) the configuration block is located in.
CRC Type	link-time	--	**CRC16** CRC32	CRC type for this block.
Ram Block	link-time	C-identifier	Valid C-identifier, default: **CfgId_RamBlock**	Symbolic name of the RAM block (Variable name as defined by application) If no name is defined, RAM Block will be defined by NvM.
Single Block Callback	link-time	C-identifier	**NULL_PTR** or other valid C-identifier	Name of the callback function supplied by application, if used. This field is disabled, if 'Use Service Ports' is checked, because in this case, the callback will be invoked, depending on RTE configuration (Port Mapping). It is also disabled, if 'Use Job End Callback' is unchecked.
Requirement Id	link-time	C-identifier	valid C-identifier	A requirement id may be entered here for traceability reason.
Priority	link-time	integer	0.. **127** ..255 (0 =immediate)	Priority of the Config ID Block **Info** Note that a priority of 0 (immediate) is possible, but not recommended.

Attribute Name	Configuration Variant	Value Type	Values The default value is written in bold	Description
RAM CRC	link-time	--	**ON** OFF	Enable/disable the calculation of CRC in RAM, upon `NvM_SetRamBlockStatus(true)` request.
Write Once	link-time	--	ON **OFF**	Configure block as write once. ⓘ **Info** When using 'Dynamic Configuration Handling', this option should be unchecked.
Initially Write Protected	link-time	--	ON **OFF**	Write protection enabled after reset.
Use Service Ports	link-time	--	**ON** OFF	Mark the configured block as accessible via service ports. The block name will be derived from the symbolic name.
Use Job End Callback	link-time	--	**ON** OFF	Enable/disable usage of the 'single block job end notification' callback
Use Job End Callback during NvM_ReadAll	link-time	--	ON **OFF**	Enable/disable usage of the 'single block job end notification' callback during NvM_ReadAll, when this block has been processed. It depends on the runtime-environment, whether this option may be enabled, or not. E.g., if the application is not yet initialized, when NvM_ReadAll is being processed, or (in case of Service Ports usage) if the RTE is not initialized yet, this option should be disabled.
Multiblock Request Status				
Blockname	link-time	C-identifier	Valid C-identifier, default: **MultiBlock Request**	Logical name of the reserved 'Multiblock request status block' (ID 0).

Attribute Name	Configuration Variant	Value Type	Values The default value is written in bold	Description
Callback	link-time	C-identifier	**NULL_PTR** or other valid C-identifier	Name of the callback function supplied by application, if used.
Include Files				
Include List	link-time	header file	Valid header file, default: **Application.h**	List of all includes, declaring the callback functions used (including the callbacks for the configuration block as well as the Multiblock Request Result Block) and the extern declarations of RAM and ROM blocks. Only files appearing in the list will be recognized. A file name in the edit filed will be ignored, unless it's added using the 'Add' button If NVRAM Blocks are configured to 'Use Service Ports' and with callbacks, you need to configure an RTE include file. It needs to be named Rte_<Service Component Name>.h

Table 7-2   Special NVRAM blocks

7.2.4   User Block Description

Attribute Name	Configuration Variant	Value Type	Values The default value is written in bold	Description
Block Description				
Identifier	link-time	integer	1.. **2**.. .. 65535	Numeric handle of the NVRAM Block to be passed as BlockId parameter to the NvM. This value will be calculated automatically. In DaVinci Configurator Pro it is not writeable by the user.

based on template version 3.01

Attribute Name	Configuration Variant	Value Type	Values The default value is written in bold	Description
Name	link-time	C-identifier	Valid C-identifier, must be unique, default: **Appl_Nvm Block1**	Logical name of the User Block, to be used within your application (Don't use the numeric Block IDs, represented by the names, directly). A block handle with this name will be generated. This name will also be used in Descriptions and long names of the NVM's SWC Description.
Device	link-time	FEE/EA or dummies	All configured NV abstraction devices (FEE/EA or dummies), **no default**.	The device the User Block is located in.
MngmtType	link-time	--	**Native** Redundant Dataset	Management Type of the User Block.
CRC Type	link-time	--	**OFF** CRC16 CRC32	CRC protection to be used for the User Block.
RAM CRC	link-time	--	ON **OFF**	If checked, the CRC of the RAM block is recalculated when `NvM_SetRamBlockStatus(TRUE)` is called.
Prio	link-time	integer	0.. **127** ..255 (0 =immediate)	Priority of the User Block.
Length	link-time	integer	**1** .. 65535	Length of the User Block in bytes.
Count	link-time	integer	**Native: 1** Redundant: 2 Dataset: 2...255	Number of mapped blocks in the NV area. Dependant on the management type.

Attribute Name	Configuration Variant	Value Type	Values The default value is written in bold	Description
Further Settings				
Service Ports	link-time	--	**ON** OFF	Generate service port description for the configured block. If checked, the parameters 'Use Init Callback', 'Init Callback' and 'Job End Callback' are disabled
WO	link-time	--	ON **OFF**	**W**rite **O**nce Flag. A write once block can only be written once. **Caution** Pay attention that a dataset block set to write once cannot be written once for all its datasets. Only one dataset could be written.
IWP	link-time	--	ON **OFF**	**I**nitial **W**rite **P**rotection for this block. Defines the state of this block's user-definable write protection after start-up.
Res	link-time	--	**ON** OFF	**Res**istant to changed software Option influences the block handling during `NvM_ReadAll()`.
ReadAll	link-time	--	**ON** OFF	If checked the block will be read from NV during `NvM_ReadAll()`.

Attribute Name	Configuration Variant	Value Type	Values The default value is written in bold	Description
Cbk @ ReadAll	link-time	--	ON **OFF**	Enable/disable usage of the 'single block job end notification' callback during NvM_ReadAll in contrast to the typical behaviour where only the 'multiblock job end notification' would be called. Nevertheless the 'multiblock job end notification' will be called after processing of all configured "ReadAll"-blocks. **Info** It depends on the runtime-environment, whether this option may be enbled, or not. E.g., if the application is not yet initialized, when NvM_ReadAll is being processed, or (in case of Service Ports usage) if the RTE is not initialized yet, this option should be disabled.
RAM Block	link-time	C-identifier	NULL_PTR or other valid C-identifier, default: **RamBlock Data1**	Symbolic name of the RAM block (Variable name as defined by application). If the block is non-permanent the field can be left empty or a NULL_PTR can be entered. **Info** A RAM block must not be allocated, if the block is configured by RTE. The field can be left empty or a NULL_PTR can be entered.

Attribute Name	Configuration Variant	Value Type	Values The default value is written in bold	Description
Service Port Settings				
ROM Block	link-time	C-identifier	**NULL_PTR** or other valid C-identifier	If the block shall have ROM defaults the name of the constant must be entered in this field (constant name as defined by application). ⚠ **Caution** Pay attention that a block can only have either ROM defaults or an init call-back. If the ROM block is configured which means that it is not empty and not NULL_PTR, the setting of the Init Callback will be ignored. ℹ **Info** A ROM block must not be allocated, if the block is configured by RTE. The field can be left empty or a NULL_PTR can be entered.

Attribute Name	Configuration Variant	Value Type	Values The default value is written in bold	Description
Use Init Callback	link-time	--	ON **OFF**	An alternative to ROM defaults is the use of an init callback. Checks this control to enable usage of the init callback. ⚠ **Caution** Pay attention that a block can only have either ROM defaults or an init call-back. If a ROM Block was configured, or if the 'Init Callback' was not set, this option does not have any effects. If 'Use Service Ports' was enabled, this option is disabled, because it cannot be guaranteed that the init callback can be invoked during NvM_ReadAll processing, as it is unclear whether the RTE is running.
Init Callback	link-time	C-identifier	**NULL_PTR** or other valid C-identifier	An alternative to ROM defaults is the use of an init callback. Enter the name in this field as defined by the aplication. This control is disabled, if 'Use Service Ports' was checked. ⚠ **Caution** Pay attention that a block can only have either ROM defaults or an init call-back. If a ROM block was configured, the 'Init Callback' will be ignored.

Attribute Name	Configuration Variant	Value Type	Values The default value is written in bold	Description
Use Job End Cbk	link-time	--	ON **OFF**	Use the Job end callback. If checked, the control 'Job End Callback' is enabled, unless 'Use Service Ports' was also checked.
Job End Callback	link-time	C-identifier	**NULL_PTR** or other valid C-identifier	Name of the callback function supplied by application, if used. ⚠ **Caution** If 'Use Service Ports' was chosen, the name of the callback is implicitly defined. Thus this control will be disabled.
Requirement Id	link-time	C-identifier	valid C-identifier	A requirement id may be entered here for traceability reason.

Table 7-3   User block description

7.2.5 Error Detection

Attribute Name	Configuration Variant	Value Type	Values The default value is written in bold	Description
Error Detection – Development Mode				
Development Error Detection	pre-compile	--	ON **OFF**	If activated then all checks – if enabled – will be done. Otherwise, no further development error detection will be done. By setting this preprocessor switch to ON the general parameter checking is enabled. **Info** In production mode, this switch should be disabled to save RAM/ROM and to speed up the module.
Check Module's Initialization Status	pre-compile	--	**ON** OFF	(De)Activate NVM initialization check within each API function call (except `NvM_Init()` and `NvM_GetVersionInfo()`).
Check Block's Management Type	pre-compile	--	**ON** OFF	(De)Activate checking of block's management type in API functions `NvM_RestoreBlockDefaults()`, `NvM_GetDataIndex()` and `NvM_SetDataIndex()` respectively.
Check Block's Write Protection	pre-compile	--	**ON** OFF	(De)Activate checking of block's write protection and also the correct use of the write protection in API functions (write once): `NvM_WriteBlock()`, `NvM_InvalidateNvBlock()`, `NvM_EraseBlock()` and `NvM_SetBlockProtection()`.
Check Block's Pending State	pre-compile	--	**ON** OFF	(De)Activate checking of block's listed/pending state in all API functions, except `NvM_GetErrorStatus()`, `NvM_GetVersionInfo()` and `NvM_Init()`.
API Parameter Checking	pre-compile	--	**ON** OFF	By setting this preprocessor switch to ON the general parameter checking is enabled.

Attribute Name	Configuration Variant	Value Type	Values The default value is written in bold	Description
Check BlockId	pre-compile	--	ON **OFF**	(De)Activate checking for block ID in correct range.
Check DataIndex	pre-compile	--	**ON** OFF	(De)Activate checking if DataIndex passed to `NvM_SetDataIndex()` is in configured range. Applies only to management type Dataset. Relevant for the functions `NvM_WriteBlock()`, `NvM_InvalidateNvBlock()` and `NvM_EraseBlock()`.
Check Pointers	pre-compile	--	**ON** OFF	(De)Activate checking if a pointer parameter is not the `NULL_PTR` (`NvM_GetDataIndex()`, `NvM_GetErrorStatus()`, `NvM_GetVersionInfo()`), or the `NULL_PTR` was passed for a non-permanent block (`NvM_ReadBlock()`, `NvM_WriteBlock()`, `NvM_RestoreRomDefaults()`).
Check RAM Block Lengths	link-time	--	**ON** OFF	(De)Activate RAM Block Length checks during initialization; see chapter 4.2.1
Check ROM Block Lengths	link-time	--	**ON** OFF	(De)Activate ROM Block Length checks during initialization; see chapter 4.2.1

Enable Development Error Reporting	pre-compile	--	**ON**
			OFF
				Enable/disable calls to DET. Nevertheless, the checks configured above will be performed, and API functions will return an appropriate value.
				By setting this preprocessor switch to ON the following options have to be configured:
				`NVM_ERRORHOOK`
				`NVM_CFG_INCLUDE_DET`

Errorhook Function	pre-compile	C-function identifier	Valid C-function identifier, default: **Det_ReportError**	Defines the function that is to be called in case of a Development Error.

Attribute Name	Configuration Variant	Value Type	Values The default value is written in bold	Description
Include File	pre-compile	header file	Valid header file, default: **Det.h**	Include this header to get the declaration of `Det_ReportError()`.
Error Detection – Production Mode				
DEM Reporting Function	pre-compile	C-function identifier	Valid C-function identifier, default: **Dem_ReportErrorStatus**	Name of the function to be called when reporting an error. If an AUTOSAR compliant DEM is used, it is named `Dem_ReportErrorStatus()`.
Include File	pre-compile	header file	Valid header file, default: **Dem.h**	Name of the header file publishing the DEM API, thus the error function named above. If an AUTOSAR compliant DEM is used, it is named `Dem.h`.

Table 7-4   Error Detection

7.2.6   Module API

7.2.6.1   Provided API group

Attribute Name	Configuration Variant	Value Type	Values The default value is written in bold	Description
Provided API	--	--	--	This group shows the API services that were currently provided by the NVRAM Manager depending on its configuration.

Table 7-5   Provided API

7.3   Attributes only configurable using GCE

The NVM contains parameters that can only be modified in GCE view; they are not visible in "Comfort View", because it is not recommended to modify them. If those parameter's differ from their default values (which are also recommended values) a warning will be issued during validation process.

Attribute Name	Configuration Variant	Value Type	Values The default value is written in bold	Description
Nvm_30_CommonVendorParams / NvmCrcPreserveValue	pre-compile	BOOL	**TRUE**/FALSE	Pre-Compile parameter, which controls CRC(32) calculation. If enabled, the calculated result does not depend on setting of "CRC Bytes per Cycle" (NvmCrcNumOfBytes). See also chapter 4.4.10
Nvm_30_CommonVendorParams / NvmCfgChkBlockLengthStrict	link-time	BOOL	**TRUE**/FALSE	Link-Time parameter, which allows you to enable/disable strict block length checks. Refer to chapter 4.2.1

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.

- NVM provides an additional customer specific API NvM_SetBlockLockStatus() for setting/resetting a lock for a NV block. (see chapter 6.4.8)

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's applications. (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 4.2.1.

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 <u>not</u> 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".
PIM	Per Instance Memory

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
EA	EEPROM Abstraction Module
ECU	Electronic Control Unit
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-informatik.com