Document Title	NV Data Handling Guideline
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	810

Document Status	published
Part of AUTOSAR Standard	Classic Platform
Part of Standard Release	R22-11

Document Change History

Date	Release	Changed by	Change Description
2022-11-24	R22-11	AUTOSAR Release Management	• No content changes
2021-11-25	R21-11	AUTOSAR Release Management	• No content changes
2020-11-30	R20-11	AUTOSAR Release Management	• Effective utilization of Block Fragmentation
2019-11-28	R19-11	AUTOSAR Release Management	• No content changes • Changed Document Status from Final to published
2018-10-31	4.4.0	AUTOSAR Release Management	• Editorial changes
2017-12-08	4.3.1	AUTOSAR Release Management	• Editorial changes
2016-11-30	4.3.0	AUTOSAR Release Management	• Initial Release

Disclaimer

Table of Contents

Table of Figures

1 Introduction and functional overview

This document gives an introduction to the basic AUTOSAR concepts on Non-volatile Memory as well as the various access mechanisms available for the application software components.

A brief outline of the Non-volatile memory concepts is provided in chapter 4 while chapter 5 provides the various use cases on accessing Non-volatile memory from an application (end user).

2 Acronyms and abbreviations

Abbreviation / Acronym:	*Description:*
NvM	NVRAM Manager
NV	Non-volatile
NVRAM	Non-volatile Random Access Memory
NVRAM Block	The NVRAM Block is the entire structure, which is needed to administrate and to store a block of NV data.
NV Block	The NV Block is a Basic Storage Object. It represents the part of a “NVRAM Block” which resides in the NV memory
RAM Block	The RAM Block is a Basic Storage Object. It represents the part of a “NVRAM Block” which resides in the RAM.
RAM Mirror	RAM mirrors are NvM internal buffer used for operations that read and write the RAM block of NVRAM blocks with NvMBlockUseSyncMechanism set TRUE.
ROM Block	The ROM Block is a Basic Storage Object. It represents the part of a “NVRAM Block” which resides in the ROM.
ROM	Read-Only Memory
RTE	Runtime Environment
SW-C	Software Component

3 Related documentation

3.1 Input documents

[1] AUTOSAR Specification for Runtime Environment
AUTOSAR_SWS_RTE.pdf

[2] AUTOSAR Template Specification of Software Component
AUTOSAR_TPS_SoftwareComponentTemplate.pdf

[3] AUTOSAR Specification for NVRAM Manager
AUTOSAR_SWS_NVRAMManager.pdf

[4] AUTOSAR Guide on Mode Management
AUTOSAR_EXP_ModeManagementGuide.pdf

3.2 Related standards and norms

None

3.3 Related specification

None

4 Overall mechanisms and concepts

4.1 NvM and its features

Changeability and durability are attributes associated with data inside an ECU. Data whose values are changeable but available across the power cycles needs to be stored in the Non-volatile memory. NV Data is that data inside the Non-volatile memory. In AUTOSAR, application can access this Non-volatile memory only via the NVRAM Manager (NvM). This module provides the required services (synchronous / asynchronous) for the management and maintenance of the data.

Figure 1 shows the interaction between applications and memory stack along with the modules involved.

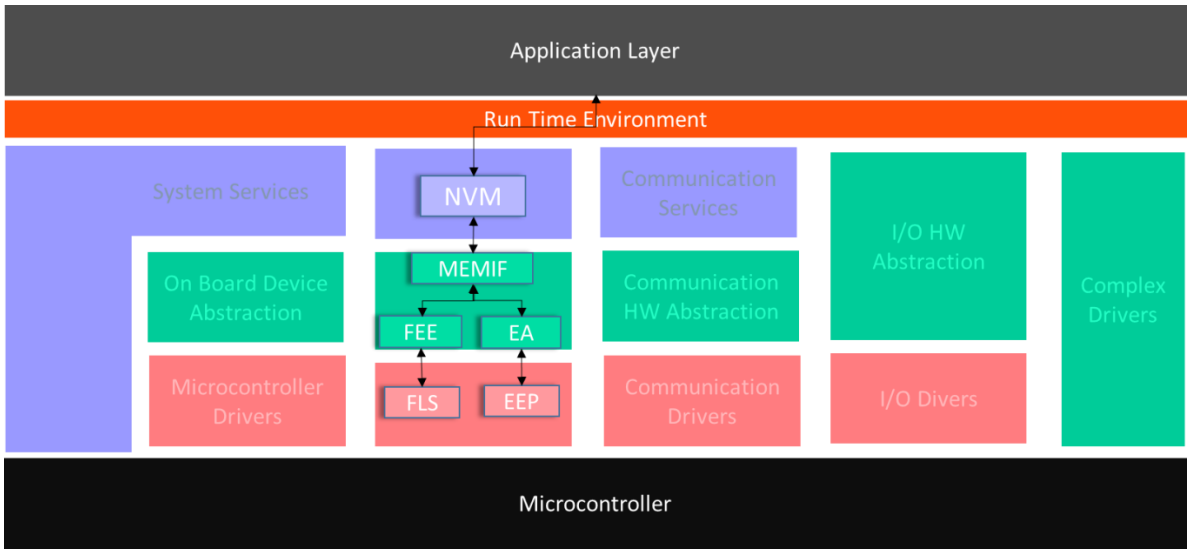


Figure 1: Overview of memory stack in AUTOSAR

Following sub sections will provide overview of different concepts and features provided by NvM module, for more details refer [3].

4.1.1 Basic storage objects

A “Basic Storage Object” is the smallest entity of a “NVRAM block”. Several “Basic Storage Objects” can be used to build a NVRAM Block. A “Basic Storage Object” can reside in different memory locations (RAM/ROM/NV memory).
These basic storage objects are described below -

4.1.1.1 RAM Block

The “RAM Block” is a “Basic Storage Object”. It represents the part of a “NVRAM Block” which resides in the RAM.

It is composed of user data and (optionally) a CRC value and (optionally) a NV block header. It is used to hold the live data. This is an optional part of NVRAM block.

4.1.1.2 ROM Block

The “ROM Block” is a “Basic Storage Object”. It represents the part of a “NVRAM Block” which resides in the ROM. The “ROM Block” is an optional part of a “NVRAM Block”.

Contents of ROM Block are of persistent nature, which can’t be modified during program execution and resides in ROM/Flash. It is used to provide default data in case of an empty or damaged NV block.

4.1.1.3 NV Block

The “NV Block” is a “Basic Storage Object”. It represents the part of a “NVRAM Block” which resides in the NV memory. The “NV Block” is a mandatory part of a “NVRAM Block”.

Contents of NV Block are of persistent nature that can be modified during program execution and resides in the Flash. It is composed of NV user data and (optionally) a CRC value and (optionally) a NV block header. It is used to hold the live data that are stored periodically/on request.

4.1.1.4 Administrative Block

The “Administrative Block” is a “Basic Storage Object”. It resides in RAM. The “Administrative Block” is a mandatory part of a “NVRAM Block”.

Contents of Administrative Block are of non-persistent nature and resides in the RAM. It is used to hold attribute/error/status information of the corresponding NVRAM blocks well as the block indices specifically for NVRAM blocks of type 'Dataset’. This is a mandatory part of NVRAM block.

4.1.2 Block Management types

The following NVRAM Block Management types are supported by the NvM:

4.1.2.1 Native NVRAM block

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

NVM_BLOCK_NATIVE type of NVRAM storage consists of the following basic storage objects:

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

4.1.2.2 Redundant NVRAM block

In addition to the Native NVRAM block, the Redundant NVRAM block provides enhanced fault tolerance, reliability and availability. It increases resistance against data corruption. The Redundant NVRAM block consists of two NV blocks, a RAM block and an Administrative block.

In case NV Block associated with a Redundant NVRAM block is deemed invalid (e.g. during read), an attempt is made to recover the NV Block using data from the incorrupt NV Block.

NVM_BLOCK_REDUNDANT type of NVRAM storage consists of the following basic storage objects:

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

4.1.2.3 Dataset NVRAM block

The Dataset NVRAM block is an array of equally sized data blocks. The application can at one-time access exactly one of this data block.

NVM_BLOCK_DATASET type of NVRAM storage consists of the following basic storage objects:

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

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

A specific dataset element is accessed by setting the corresponding index using the API NvM_SetDataIndex. Elements with an index from 0 up to NvMNvBlockNum - 1 represent the NV Blocks, while the ones with an index from NvMNvBlockNum up to NvMNvBlockNum + NvMRomBlockNum - 1 represent the ROM blocks. The NVRAM Block user has to ensure that a valid dataset index is selected before accessing data elements.

4.1.3 Synchronization Mechanism supported

Two types of synchronization mechanisms are supported while accessing data to and from NvM module's RAM mirror.

4.1.3.1 Implicit synchronization

In the Implicit synchronization, Application and NvM have concurrent access to a common RAM Block. Application writes/reads the data to/from RAM by invoking NvM API's.

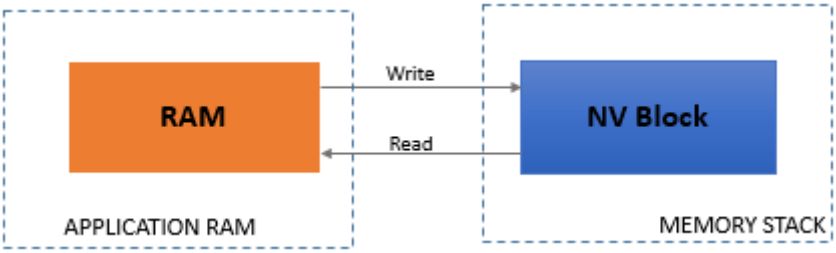


Figure 2: Overview of Implicit synchronization

In this case, RAM Block is mapped to one SW-C and sharing of RAM block is not recommendable. Whenever SW-C accesses NVRAM using RAM block (temporary / permanent), it has to ensure the data consistency of the RAM block until ongoing operation is completed by the NvM.

Following steps need to be considered while using Implicit synchronization.

- **Write request:**

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

- **Multi block write request (NvM_WriteAll):**

  1. The ECU state manager issues the *NvM_WriteAll* request which transfers control to the NvM module.
  2. The ECU state manager can use polling to get the status of the request or can be informed via a callback function.

4.1.3.2 Explicit synchronization

In Explicit synchronization, NvM defines a RAM mirror which is used to exchange data with the RAM block of Application. Application writes the data in RAM block and invokes NvM write API. NvM invokes API to read the RAM mirror and data is copied from RAM mirror to RAM block and finally to NV block. The data is transferred by the application in both directions via callback routines, called by the NvM module.

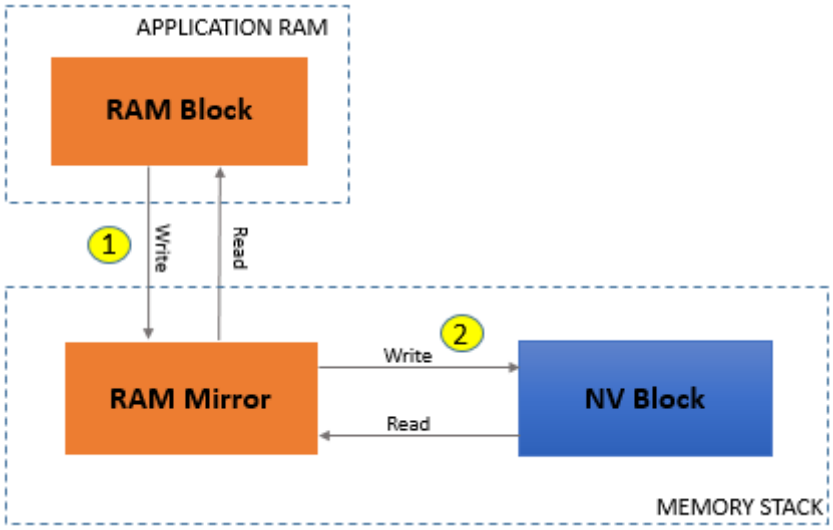


Figure 3: Overview of Explicit synchronization

The advantage is that applications can control their RAM block in an efficient way. They are responsible for copying consistent data to and from the NvM module's RAM mirror using *ReadRamBlockFromNvM* / *WriteRamBlockToNvM*. Application has to ensure data integrity of RAM block while copying data to/from RAM mirror.

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

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

Following steps need to considered while using Explicit synchronization

- **Write request:**

1. The application fills a RAM block with the data that has to be written by the NvM module.
2. The application issues the *NvM_WriteBlock* or *NvM_WritePRAMBlock* request.
3. The application might modify the RAM block until the routine *NvMWriteRamBlockToNvM* is called by the NvM module.
4. If the routine *NvMWriteRamBlockToNvM* is called by the NvM module, then the application has to provide a consistent copy of the RAM block to the destination requested by the NvM module. The application can use the return value E_NOT_OK in order to signal that data was not consistent. The NvM module will accept this *NvMRepeatMirrorOperations* times and then postpones the request and continues with its next request.
5. Continuation only if data was copied to the NvM module:
6. From now on the application can read and write the RAM block again.
7. An application can use polling to get the status of the request or can be informed via a callback routine asynchronously.

- **Multi block write request (NvM_WriteAll):**

1. The ECU state manager issues the *NvM_WriteAll* request which transfers control to the NvM module.
2. During *NvM_WriteAll* job, if a synchronization callback (*NvM_WriteRamBlockToNvM*) is configured for a block it will be called by the NvM module. In this callback the application has to provide a consistent copy of the RAM block to the destination requested by the NvM module. The application can use the return value E_NOT_OK in order to signal that data was not consistent. The NvM module will accept this *NvMRepeatMirrorOperations* times and then report the write operation as failed.
3. Now the application can read and write the RAM block again.
4. The ECU state manager can use polling to get the status of the request or can be informed via a callback function.

4.1.4 Other features

4.1.4.1 CRC based comparison

The NvM module internally uses CRC generation routines (8/16/32 bit) to check and to generate CRC for NVRAM blocks as a configurable option.

The NvM module provides an option to skip writing of unchanged data by implementing a CRC based compare mechanism. CRC based compare mechanism can be enabled by setting configuration parameter *NvMBlockUseCRCCompMechanism*.

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

4.1.4.2 Error recovery

The NvM module provides implicit error recovery on read for NVRAM block management types NATIVE and REDUNDANT by loading default values (if configured via either the parameter *NvMRomBlockDataAddress* or the parameter *NvMInitBlockCallback*).

The explicit retrieval of ROM data is available for all block management types by calling the API NvM_RestoreBlockDefaults. For DATASET, the related index must be set (pointing at a ROM block) prior to calling this API.

The NvM module provides error recovery on read for NVRAM blocks of block management type *NVM_BLOCK_REDUNDANT* by loading the RAM block with default values.

The NvM module provides error recovery on write by performing write retries regardless of the NVRAM block management type.

4.1.4.3 Write Verification

In case of write verification, when a RAM Block is written to NV memory, the NV block is immediately read back and compared with the original content in RAM Block.

If the original content in RAM Block is not the same as read back, then write retries are performed. And if enabled, the production code error *NVM_E_VERIFY_FAILED* is reported to DEM.

If the read back operation fails, then read retries are not performed.

4.1.4.4 RAM Block handling using the NvM_SetRamBlockStatus API

4.1.4.4.1 During startup phase (NvM_ReadAll)

For some NVRAM Blocks it might be required to preserve the data contents of the respective RAM Block from being overwritten during *NvM_ReadAll*, in case the data stored in the respective NV Block is older than the one in the RAM Block (e.g. in case of a warm reset when the data in RAM was not yet written to NV memory). In such a case the RAM Block has to be allocated in a reset-safe (non-initialized) RAM area and the configuration parameter *CalcRamBlockCrc* has to be set to TRUE (this implies that the corresponding NV block(s) also has/have a CRC configured) and the parameter *NvMSetRamBlockStatusApi* has to be set to the value TRUE.

After every change of the RAM Block data content the API *NvM_SetRamBlockStatus* has to be called for the corresponding NVRAM Block, with the parameter BlockChanged set to TRUE. The NVRAM Manager will then recalculate the CRC of this RAM Block and store the result in an internal variable allocated in the reset-safe (non-initialized) RAM area. As a prerequisite for this NVRAM block either a valid permanent RAM Block (*NvMRamBlockDataAddress*) or an explicit synchronization callback function (*NvMReadRamBlockFromNvM*) has to be configured.

During every startup (*NvM_ReadAll*) the NvM module calculates the CRC over such a RAM Block and if it matches with the stored CRC value the RAM Block will not be overwritten. In case the calculated CRC does not match with the stored one, the RAM Block will be overwritten with data read from the NV Block or, if this read attempt fails, with default data (if configured via either the parameter *NvMRomBlockDataAddress* or the parameter *NvMInitBlockCallback*).

4.1.4.4.2 During shutdown phase (NvM_WriteAll)

In case the configuration parameter *NvMSetRamBlockStatusApi* is set to the value FALSE, the NVRAM Manager copies during the *NvM_WriteAll* process the data content of the RAM Block to the corresponding NV Block for all NVRAM Blocks configured for WriteAll (configuration parameter *NvMSelectBlockForWriteAll* is set to the value TRUE) and having a permanent RAM Block (*NvMRamBlockDataAddress*) or an explicit synchronization callback function (*NvMReadRamBlockFromNvM*) configured.

In order to minimize the number of write cycles to the NV memory it is useful to only copy the content of those RAM Blocks to the corresponding NV Blocks, for which the data content has been changed by the NVRAM Block user. In order to enable this feature during the *NvM_WriteAll* process the configuration parameter *NvMSetRamBlockStatusApi* has to be set to the value TRUE. In this case the NVRAM

Block user has to inform the NVRAM Manager after every change made in the RAM Block data the has to be informed by calling the API *NvM_SetRamBlockStatus* for the corresponding NVRAM Block, with the parameter BlockChanged set to TRUE. This way this NVRAM Block will be marked as to be processed during the *NvM_WriteAll* process.

4.1.4.5 Resistant to changed software

The behavior of the NvM module during start-up (i.e. while processing the request NvM_ReadAll) will be influenced by the two configuration parameters NvMDynamicConfiguration and NvMResistantToChangedSw.

In ECU projects, in which it is not important to react on configuration changes of the NVRAM Blocks, the parameter *NvMDynamicConfiguration* has to be set to FALSE. For NVRAM Blocks with the configuration parameter *NvMCalcRamBlockCrc* set to TRUE, the assigned RAM block is then checked for its validity. If the RAM block content is detected to be invalid, or if the parameter *NvMCalcRamBlockCrc* set to FALSE, the NV block is checked for its validity. A NV block, which is detected to be valid, is copied to its assigned RAM block. If an invalid NV Block is detected default data will be loaded (if configured via either the parameter *NvMRomBlockDataAddress* or the parameter *NvMInitBlockCallback*).

In case the configuration of NVRAM Blocks was changed, while the NV Blocks already stored in NV Memory still correspond to the old configuration, critical problems may arise during the *NvM_ReadAll* process. An example for this is when a new NVRAM Block was added, the identifier of many other blocks might implicitly be changed, which could lead to wrong data read from the NV memory.

For such situations, it is possible to configure the NvM module so that it will not try to initialize RAM Blocks with data from NV memory. This has to be done by setting the value of the configuration parameter *NvMDynamicConfiguration* to TRUE. A change of the NVRAM configuration has to be indicated to the NvM module by modifying the configuration parameter *NvmCompiledConfigID* by the integrator. The NvM module stores this value in NV memory, using a separate NVRAM Block. With every execution of the start-up process (*NvM_ReadAll*) the NvM module compares the value stored in NV memory with the value of the configuration parameter *NvmCompiledConfigID*. In case both values are different the value in the NV memory will be overwritten by the one from the configuration with the next shutdown process (*NvM_WriteAll*).

In such a situation, there are two different possibilities how the NvM module will initialize the NVRAM Blocks during the processing of *NvM_ReadAll*, based on the value of the corresponding configuration parameter *NvMResistantToChangedSw*.

- If the data of the respective NV block shall be ignored and instead default data (if configured via either the parameter *NvMRomBlockDataAddress* or the parameter *NvMInitBlockCallback*) shall be loaded, independent of the validity of an assigned RAM block, the value of the configuration parameter *NvMResistantToChangedSw* has to be set to FALSE.
- For the NVRAM Blocks, that need to have the RAM Block to be initialized with the data from NV memory, even in case of a configuration change, the

configuration parameter *NvMResistantToChangedSw* has to be set to TRUE. The behavior will be the same as when no configuration change occurs.

For blocks having *NvMResistantToChangedSw* set to TRUE the integrator has to ensure that the following configuration parameters must not be changed for the rest of the ECU's lifetime, because otherwise it will not be possible to successfully retrieve the data from NV memory:

- NvMResistantToChangedSw (must not be changed from TRUE to FALSE)
- ShortName
- NvMBlockUseCrc
- NvmBlockCrcType (if NvMBlockUseCrc is set to TRUE)
- NvMStaticBlockIDCheck
- NvmNvramDeviceId
- NvmBlockManagementType
- NvmNvBlockLength
- NvmNvBlockBaseNumber

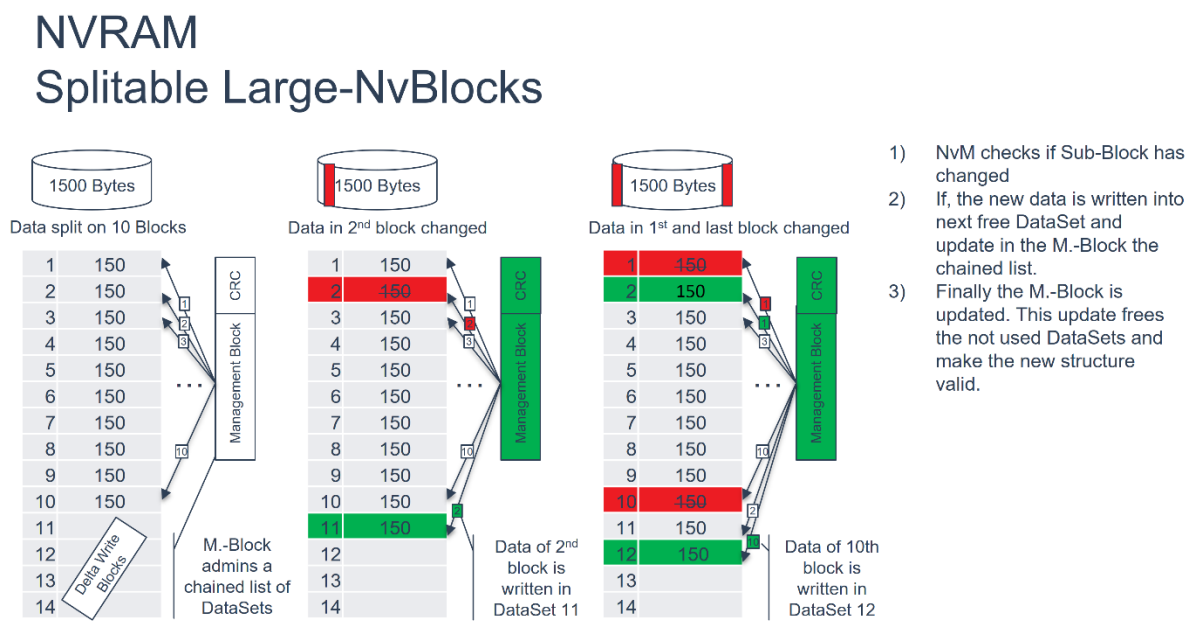
Note: Additional constraints may apply depending on the implementation of the used modules NvM, Fee and Ea. Please refer to the respective user manuals.

4.1.4.6 Block Fragmentation

Currently, there is an increasing demand for huge data blocks that have to be stored in NvRAM. These data are typically organized in huge arrays, but the update per driving cycle is usually local. Such an array is usually organized only in a single NvBlock. A more frequent update, e.g. after each driving cycle, shortens the lifetime of the ECU or requires the use of a larger DataFlash.

This feature offers the possibility of fragmenting a large block into smaller units. These fragments shall only be written if the content has been changed. The unchanged fragments shall be kept unchanged.

The following picture shows the concept and the evolution of the NvRAM dataset mapping in case of local / spare updated data inside an array data element object.



Note: The fragmentation can be implemented efficiently only when the NVM elements are grouped according to the write frequency. The NVM elements shall be arranged in such a way that they are in either increasing or decreasing order of write frequency.

4.2 Accessing NvM using RTE

Details of possible Interfaces, Software component types to access NV data using RTE are listed in this section, for more details refer [1] and [2].

4.2.1 Interfaces

- **Client-server interface**
  A client/server interface provides a number of operations that can be invoked on a server by a client. In case of services provided by NvM to the Application, NvM acts as server and Application acts as client. Also notifications to be provided to the Applications from NvM can be implemented using same; role of client and server is exchanged in this case.
- **NvDataInterface**
  A non-volatile data interface defines a number of *VariableDataPrototypes* to be exchanged between non-volatile block components and atomic software components. These *VariableDataPrototypes* can be mapped to complete RAM block or elements of RAM block implemented inside non-volatile block components.

4.2.2 Accessing NV Data using ServiceSwComponent

NvM is configured as ServiceSwComponent. Here SW-C(s) which wants to read/write data to NVRAM needs to utilize the standard NvM services using a Client-Server Interface.

Refer ***Use case 1*** and ***Use case 2*** for more details on using NvM as ServiceSwComponent.

- **Benefits:**
  - Enables basic generic configuration to accommodate NvM services and callbacks using the SW-C template.
  - Dedicated set of ports available for each block.
  - Application is abstracted from Block Identifier of NVRAM Block allocated by NvM.

- **Restoring Default Value**
  If Application is maintaining the RAM Blocks, it can define ParameterDataPrototypes local to a SW-C using PerInstanceParameter or ConstantMemory configuration which can be used by the SW-C to restore the RAM block and by NvM to restore the NV block.

- **Handling Notifications**
  Notifications from NvM to the Application are implemented by the RTE using a Client-Server Interface in the scenario of which NvM acts as the client and the user of the NvBlock acts as the server.

  In case of NvM being used via a ServiceSwComponent, the NvM uses generic Client-Server API mapped to a r-port of the NvM ServiceSwComponent i.e. Rte_Call() which is connected to the p-port of the user of the NvBlock (SW-C).

4.2.3 Accessing NV Data using NvBlockSwComponent

Here NvM is configured as NvBlockSwComponent in RTE and can be utilized to create RAM Blocks (mirrors) of its own which can be written/read partially or completely by a single or multiple SW-C's using a NV-Data Interface.

Along with this, the Client-Server Interfaces are used for the NvM services and notifications.

Refer ***Use case 3*** for more details on using NvM as NvBlockSwComponent.

- **Benefits:**
  - Each block is identified and has a dedicated RAM block allocated by the RTE.
  - Application SW-C implementation are independent of the name and type of the RAM block.
  - RAM block can be shared between multiple SW-C's via partial data mapping mechanisms provided by RTE (PortInterfaceMapping, NvDataMapping).
  - Less memory is used to implement RAM blocks.

- Application always access RAM blocks via port interfaces resulting into a more modular approach.
- User can utilize the dirtyFlag mechanism to enable a writing strategy in the NvBlockSwComponent instead of implementing a writing strategy in the Application software.

- **Restoring Default Value**
  NvBlockDescriptor inside a NvBlockSwComponent allows configuration of optional ROM blocks using *ParameterDataPrototype*. This ROM block constant is used as an *initValue* to which the RAM block can be restored to after initialization of the RTE or a partition.

- **Handling Notifications**
  Notifications from NvM to the Application are implemented by the RTE using a Client-Server Interface in the scenario of which NvM acts as the client and the user of the NvBlock acts as the server.

  In case of NvM being used via a NvBlockSwComponent, the NvM uses standard callback API's provided by RTE for each *NvBlockDescriptor* having *RoleBasedPortAssignment* for a ClientServerPort (r-port) with a *NvMNotifyJobFinished* or *NvMNotifyInitBlock* role i.e. *Rte_NvMNotifyInitBlock*() and *Rte_NvMNotifyJobFinished*().

4.3 Initialization of RAM Blocks from NVRAM

There are different strategies in AUTOSAR to restore RAM blocks to their previous values i.e. the values held before going into last shutdown.

Individual blocks can be explicitly read one-by-one using *NvM_ReadBlock/NvM_ReadPRAMBlock* from the Initialization runnables called using InitEvent during *Rte_Init().*

A more optimized approach is to read all such blocks where data persistence is required using a single NvM request *NvM_ReadAll.* All such NvM blocks are configured to be applicable for NvM_ReadAll. Any block to be read during *NvM_ReadAll* has to either have explicit synchronization or have a permanent RAM block. Figure 4 shows the process of restoration of RAM block using *NvM_ReadAll*. Further information about the AUTOSAR startup process is mentioned in [4].

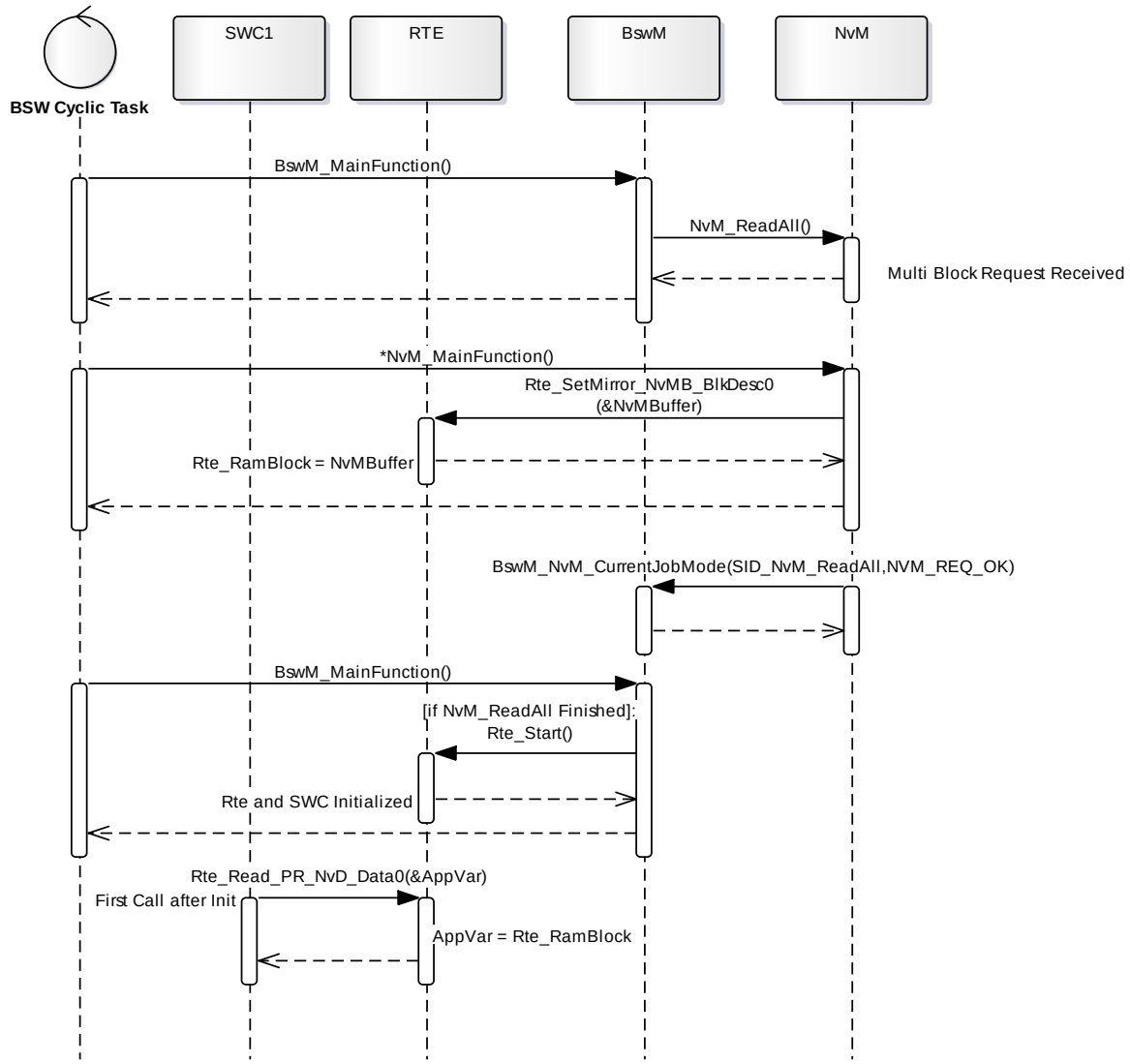


Figure 4: Sequence diagram for Initialization of RAM block

5 Use Case Summary

The use-cases are segregated with respect to allocation of RAM blocks by Application or the RTE. In the majority of use-cases, all of the services are implemented by the RTE and used by the Application.

It is assumed that the RAM blocks have been initialized from the NVRAM on startup using one of the possible strategies within AUTOSAR elaborated in section 4.3.
This use cases are applicable to all types of NVRAM blocks specified in section 4.1.2.

Following table provides overview of all use cases described in the below sub-sections:

Section	***Title***	***Description***
5.1	Application SW-C accessing NVRAM Blocks having no Permanent RAM block	In this use case, SW-C is responsible for allocating RAM blocks that are used to access NVRAM block via Client-server ports.
5.2	Application SW-C accessing NVRAM blocks which have Permanent RAM blocks	In this use case, the RAM block is allocated in the RTE using PerInstanceMemory and is used to access NVRAM block via Client-Server ports.
5.3	Application SW-C accessing NVRAM block using an NvBlockSwComponentType	In this scenario, the RTE allocates the RAM Blocks according to the definition in the NvBlockSwComponent and those are then partially/completely written/read by a single or multiple SW-C's using a NV-Data Interface.

5.1 Case 1: Application SW-C accessing NVRAM Blocks having no Permanent RAM block

In all the scenarios covered by the use-case, NvM is configured in the form of a ServiceSwComponent. Any Application SW-C which wants to read/write data to NVRAM needs to utilize the standard NvM services using a Client-Server Interface.

Set of ports can be uniquely mapped to a NvMBlockDescriptor using the Port Defined Argument Value to determine the Identifier of the NvBlock.

The above use case is categorized further into two scenarios depending on the type NvM Block Synchronization used:

5.1.1 Case 1a: Application providing reference to its RAM data area

Implicit Synchronization is configured for the particular NvMBlockDescriptor for this use case (*NvMBlockUseSyncMechanism* parameter set to false). This mechanism is also known as "Using Temporary RAM Block by NvM".

In this scenario, the Application provides the reference of RAM data area as an argument as part of the *NvM_ReadBlock* / *NvM_WriteBlock* API's. Thus, it is the responsibility of the User (SW-C) to ensure data consistency for the RAM data (e.g. writing into the RAM data area while NvM is using it).

NvMRamBlockDataAddress parameter is not configured for this use case.

Figure 5 provides an overview of memory allocation in Application and NvM as per this use case. For details about interaction between these modules refer sequence diagram in figure 7.

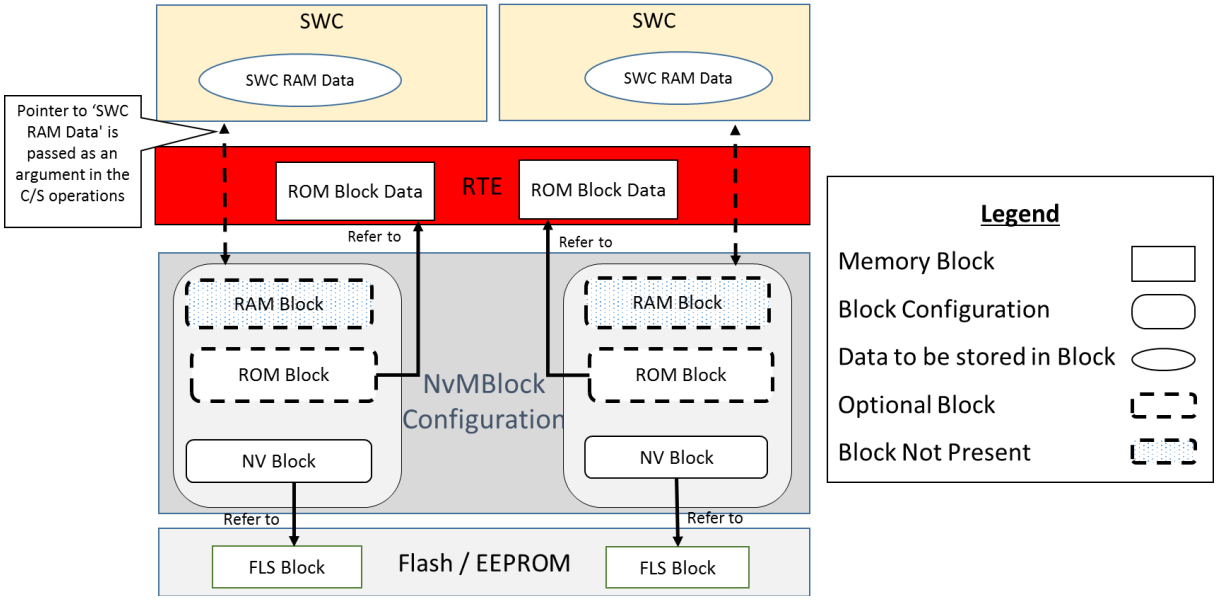


Figure 5: Overview of memory allocation for use case 1a

Figure 6 shows port configuration used to access NvM interfaces by user (SWC1 / SWC2), here NvM is configured as ServiceSwComponentType.

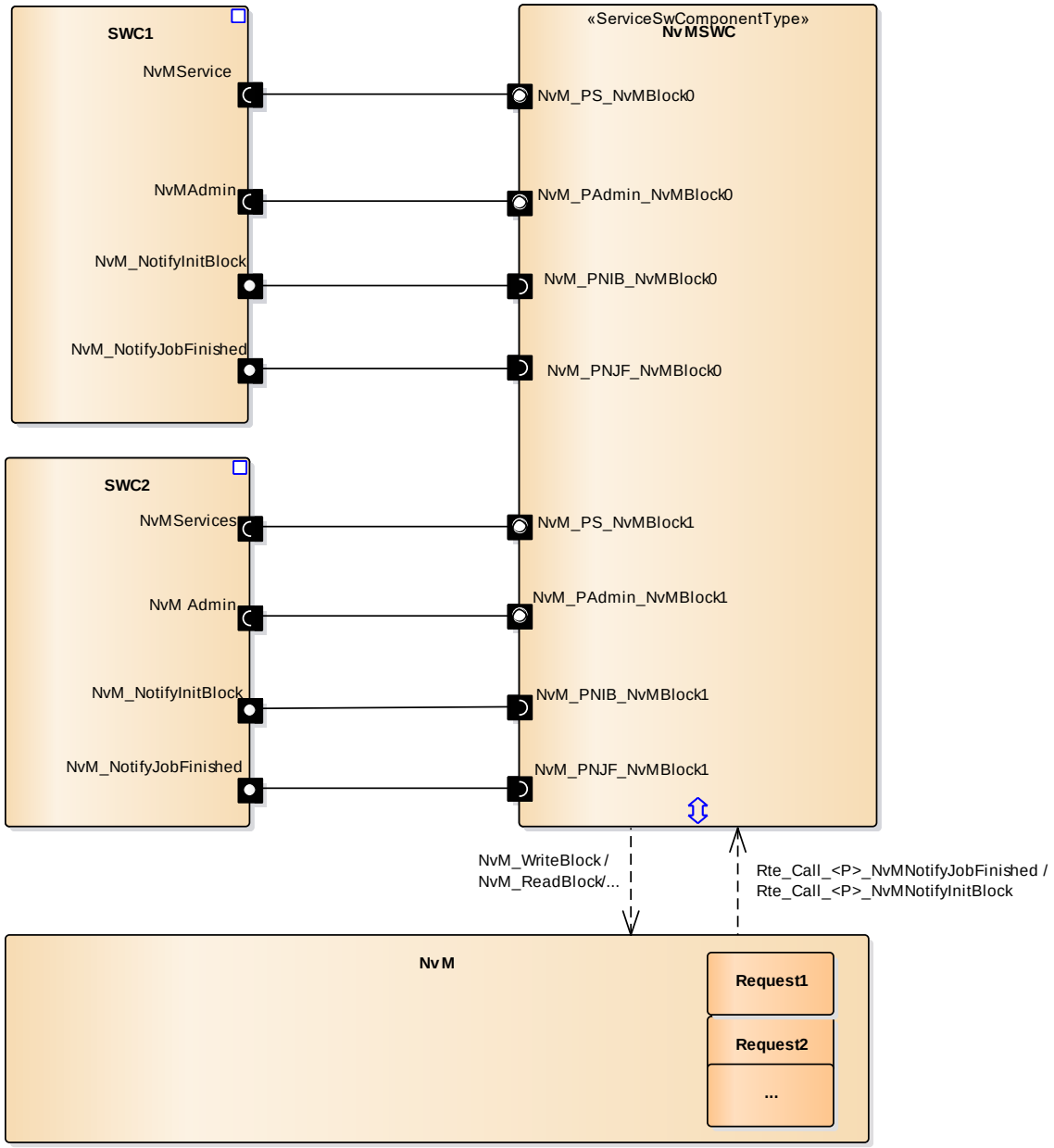


Figure 6: Port configuration diagram for use case 1a

Following ports are required to be configured –

- NvMService interface is used to send commands to the NvM. Some of the basic operations supported are:
  - *EraseBlock*
  - *GetDataIndex*
  - *GetErrorStatus*
  - *InvalidateNvBlock*
  - *ReadBlock*
  - *RestoreBlockDefaults*
  - *SetDataIndex*
  - *SetRamBlockStatus*
  - *WriteBlock*

Note: - Services provided by NvM are configurable as per the “NVRAM Manager API configuration classes” specified in [3].

- NvMAdmin interface is used to order some administrative operations to the NVM. Following operation is supported.
  - *SetBlockProtection*
- NvM_NotifyInitBlock interface is used by the NVM to request users to provide the default values. Following operation is supported.
  - *InitBlock*
- NvM_NotifyJobFinished interface is used by the NVM to notify the end of job. Following operation is supported.
  - *JobFinished*

Figure 7 shows how application can access NvM in this scenario:

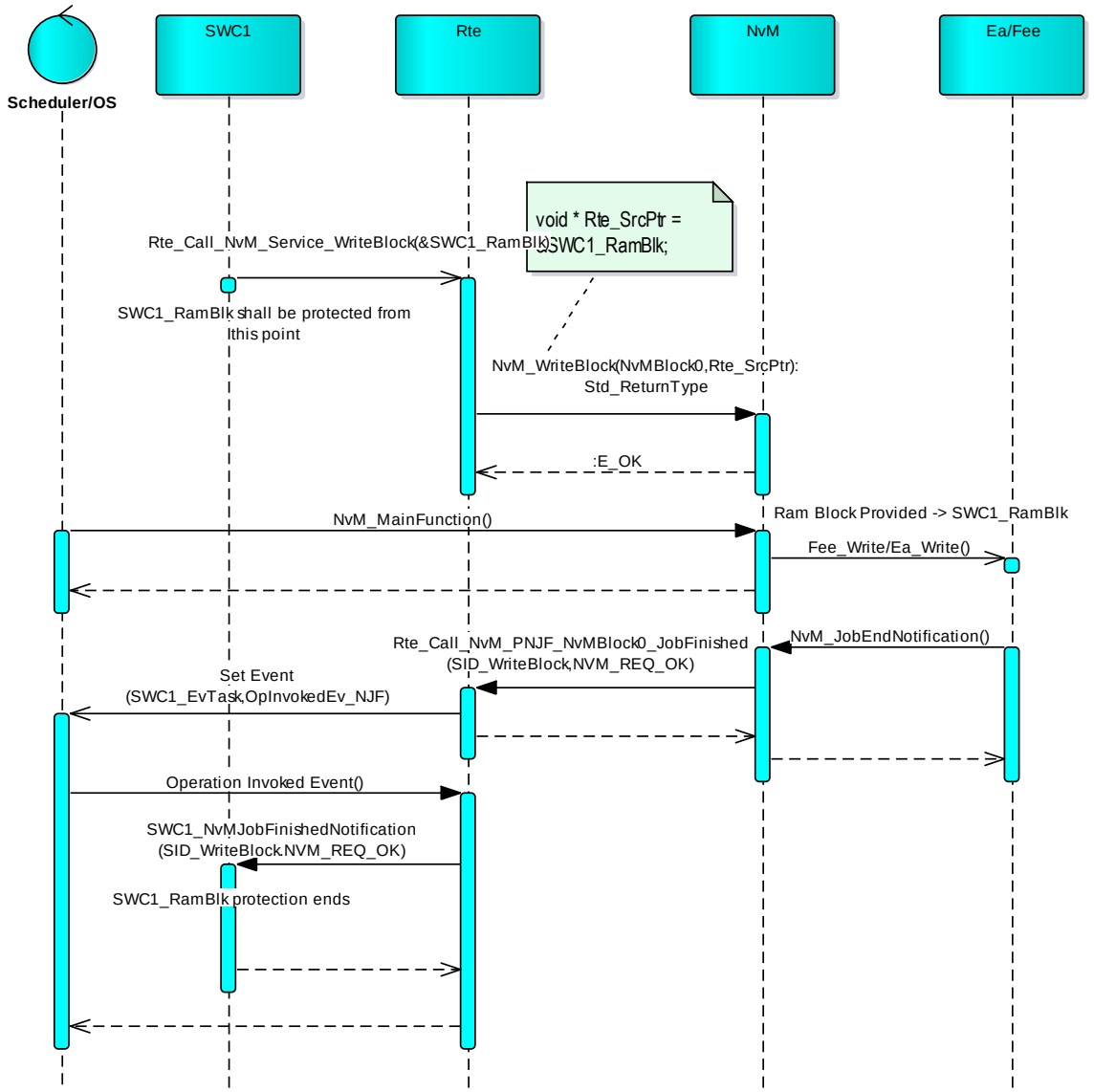


Figure 7: Sequence diagram for NvM access for use case 1a

- **Applicability:**

  In this use case, RAM Blocks are not present and thus user (SW-C) have to be synchronized to guarantee that no unsuitable accesses to the RAM data take place during NVRAM operations.

  This use case is applicable to scenarios where NvData mapped to NvMBlock are not shared between different SW-C's. Application has to ensure data consistency while accessing application RAM data area concurrently.

  For Data consistency, application should not access application RAM data provided to the NvBlock after call to *NvM_ReadBlock* / *NvM_WriteBlock* until it has been notified - via the JobFinished callback - that the job was successful.

- **Configuration:**
  Refer below complete configuration prepared for this use case.
  Configuration files –
  ConfigurationFiles/NvDataHandling_*ServiceSwComponent_ImplicitSynchronization/*.arxml*

Details of configuration applicable to this use case is as shown below:

Configuration for SWC1:

Interface		NvMService	
Client	SWC1	Server	NvM
R-Port	NvM_Service	P-Port	NvM_PS_NvMBlock0
Synchronous Server CallPoints	WriteBlock, ReadBlock, RestoreBlockDefaults, GetErrorStatus, SetRamBlockStatus, Get/SetDataIndex etc.	Operation Invoked Events	WriteBlock, ReadBlock, RestoreBlockDefaults, GetErrorStatus, SetRamBlockStatus, Get/SetDataIndex etc.

Interface		NvMAdmin	
Client	SWC1	Server	NvM
R-Port	NvM_Admin	P-Port	NvM_PAdmin_NvMBlock0
Synchronous Server CallPoints	SetBlockProtection	Operation Invoked Events	SetBlockProtection

Interface		NvMNotifyInitBlock	
Client	NvM	Server	SWC1
R-Port	NvM_PNIB_NvMBlock0	P-Port	NvM_NotifyInitBlock
Synchronous Server CallPoints	NotifyInitBlock	Operation Invoked Events	NotifyInitBlock

Interface		NvMNotifyJobFinished	
Client	NvM	Server	SWC1
R-Port	NvM_PNJF_NvMBlock0	P-Port	NvM_NotifyJobFinished

Synchronous Server CallPoints	NotifyJobFinished	Operation Invoked Events	NotifyJobFinished

Configuration for SWC2:

Interface		NvMService	
Client	SWC2	Server	NvM
R-Port	NvM_Service	P-Port	NvM_PS_NvMBlock1
Synchronous Server CallPoints	WriteBlock, ReadBlock, RestoreBlockDefaults, GetErrorStatus, SetRamBlockStatus, Get/SetDataIndex etc.	Operation Invoked Events	WriteBlock, ReadBlock, RestoreBlockDefaults, GetErrorStatus, SetRamBlockStatus, Get/SetDataIndex etc.

Interface		NvMAdmin	
Client	SWC2	Server	NvM
R-Port	NvM_Admin	P-Port	NvM_PAdmin_NvMBlock1
Synchronous Server CallPoints	SetBlockProtection	Operation Invoked Events	SetBlockProtection

Interface		NvMNotifyInitBlock	
Client	NvM	Server	SWC2
R-Port	NvM_PNIB_NvMBlock1	P-Port	NvM_NotifyInitBlock
Synchronous Server CallPoints	NotifyInitBlock	Operation Invoked Events	NotifyInitBlock

Interface		NvMNotifyJobFinished	
Client	NvM	Server	SWC2
R-Port	NvM_PNJF_NvMBlock1	P-Port	NvM_NotifyJobFinished
Synchronous Server CallPoints	NotifyJobFinished	Operation Invoked Events	NotifyJobFinished

5.1.2 Case 1b: NvM fetches application RAM data via callback (NvM Explicit Synchronization)

NvM Block Descriptor is required to be configured to have explicit synchronization for this use case (*NvMBlockUseSyncMechanism* parameter set to true).

In this scenario, data is maintained locally by the application SW-C. A call to *NvM_WriteBlock* will start the process but the data can be modified till that point in time when a callback referred by configuration parameter *NvMWriteRamBlockToNvCallback* occurs. The data is copied from the application data area to the NvM Ram Mirror as part of this callback. Similarly, during read operation, application data area is updated from NvM module's mirror via callback configured using *NvMReadRamBlockFromNvCallback* parameter.

This is similar to a deferred operation where unlike earlier scenarios, an SW-C can modify its data after a call to *NvM_WriteBlock* even when it is not yet written to an NV Block, as data area is accessed only during callback (protection needs to be ensured during the callback).

The callbacks are implemented using a Client Server Interface specified in 4.2.

Figure 8 provides an overview of memory allocation in Application and NvM as per this use case. For details about interaction between these modules refer sequence diagram in figure 10.

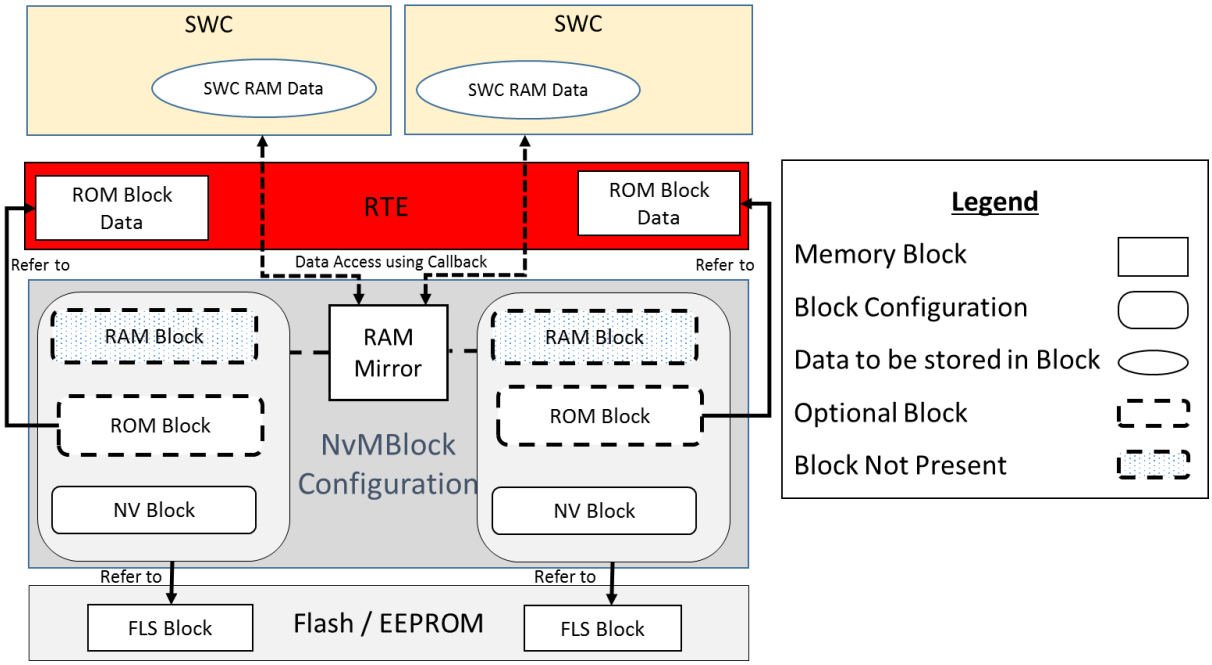


Figure 8: Overview of memory allocation for use case 1b

Figure 9 shows port configuration used to access NvM interfaces by user (SWC1 / SWC2), here NvM is configured as ServiceSwComponentType.

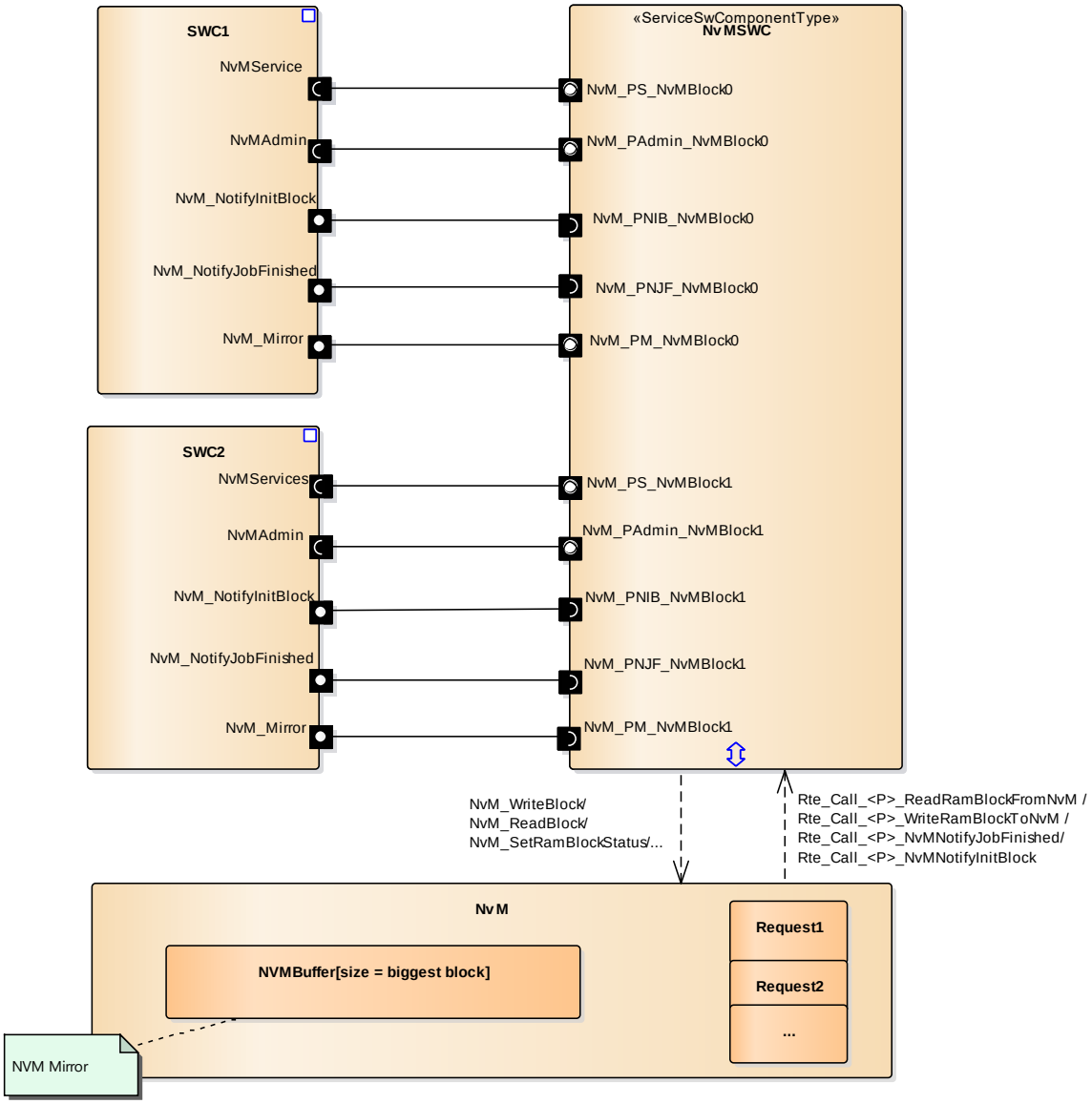


Figure 9: Port configuration for use case 1b

Refer section 5.1.1 for details of operation for ports NvMService, NvMAdmin, NvM_NotifyInitBlock and NvM_NotifyJobFinished.

Following ports are required –

- NvM_Mirror interface used to update data from NvM Mirror to application RAM data or from application RAM data to NvM Mirrors. Following operations are supported.
  - *ReadRamBlockFromNvM*
  - *WriteRamBlockToNvM*

Figure 10 shows how application can access NvM in this scenario:

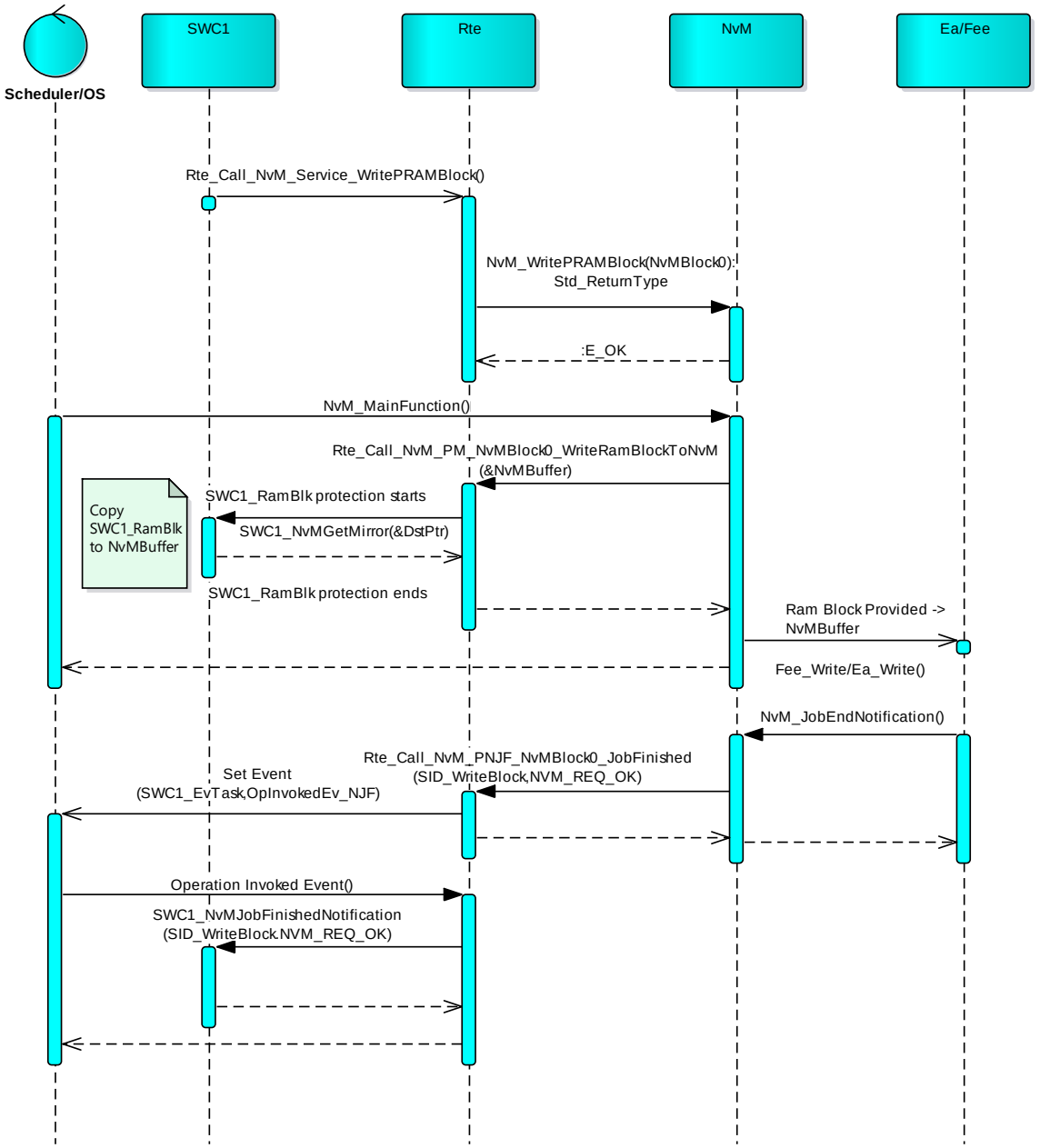


Figure 10: Sequence diagram of NvM access for use case 1b

- **Applicability:**
  In this use case, NvM users (SW-C) can control their data in a more efficient way as they are responsible for copying consistent data to and from the NvM module's RAM mirror. It is possible to modify application data just before writing into RAM mirror.

  The drawback is need of an additional RAM in NvM module that needs to have the same size as the NVRAM block and the necessity of an additional copy for read / write operation.

- **Configuration:**

  Refer below complete configuration prepared for this use case
  Configuration file –
  ConfigurationFiles/
  NvDataHandling_ServiceSwComponent_ExplicitSynchronization /.arxml*

Configuration for service ports NvMService, NvMAdmin, NvM_NotifyInitBlock and NvM_NotifyJobFinished is the same as provided in section 5.1.1.
Refer below configuration for NvM_Mirror:

For SWC1:

Interface		NvMMirror	
Client	NvM	Server	SWC1
R-Port	NvM_PM_NvMBlock0	P-Port	NvM_Mirror
Synchronous Server CallPoints	SetMirror, GetMirror	Operation Invoked Events	SetMirror, GetMirror

For SWC2:

Interface		NvMMirror	
Client	NvM	Server	SWC2
R-Port	NvM_PM_NvMBlock1	P-Port	NvM_Mirror
Synchronous Server CallPoints	SetMirror, GetMirror	Operation Invoked Events	SetMirror, GetMirror

5.2 Case 2: Application SW-C accessing NVRAM blocks which have Permanent RAM blocks

In this scenario, RAM blocks are configured in the form of Per Instance Memory in RTE. The reference to the created Per Instance Memory is given in the NvMBlockDescriptor configuration using the *NvMRamBlockDataAddress* parameter.

NvM Block Descriptor is configured to have implicit synchronization for this use case (*NvMBlockUseSyncMechanism* parameter set to false). In case of permanent RAM blocks configured in NvM, NvM provides dedicated API's to request services regarding the Permanent RAM blocks i.e. *NvM_ReadPRAMBlock/NvM_WritePRAMBlock.*

Additionally, if the User has a need to provide temporary RAM blocks even though a permanent one is configured in the system, *NvM_ReadBlock/NvM_WriteBlock* API is used to provide the temporary RAM block address. Such API's have precedence over *NvM_ReadPRAMBlock/NvM_WritePRAMBlock*.

The (ArTyped)PerInstanceMemory is allocated as RAM block in the RTE and can be accessed by the Application SW-C using *Rte_Pim* API. In this scenario, it is invalid for two SW-C's to use the RAM block for their purpose.

In general, there are two different kinds of Per Instance Memory available which are varying in the typing mechanisms. ‘C’ typed PerInstanceMemory is typed by the description of a ‘C’ typedef whereas arTypedPerInstanceMemory (AUTOSAR Typed Per Instance Memory) is typed by the means of an AutosarDataType. Nevertheless, both kinds of Per Instance Memory are accessed via the Rte_Pim API.

Figure 11 provides an overview of memory allocation in Application and NvM as per this use case. For details about interaction between these modules refer sequence diagram in figure 13.

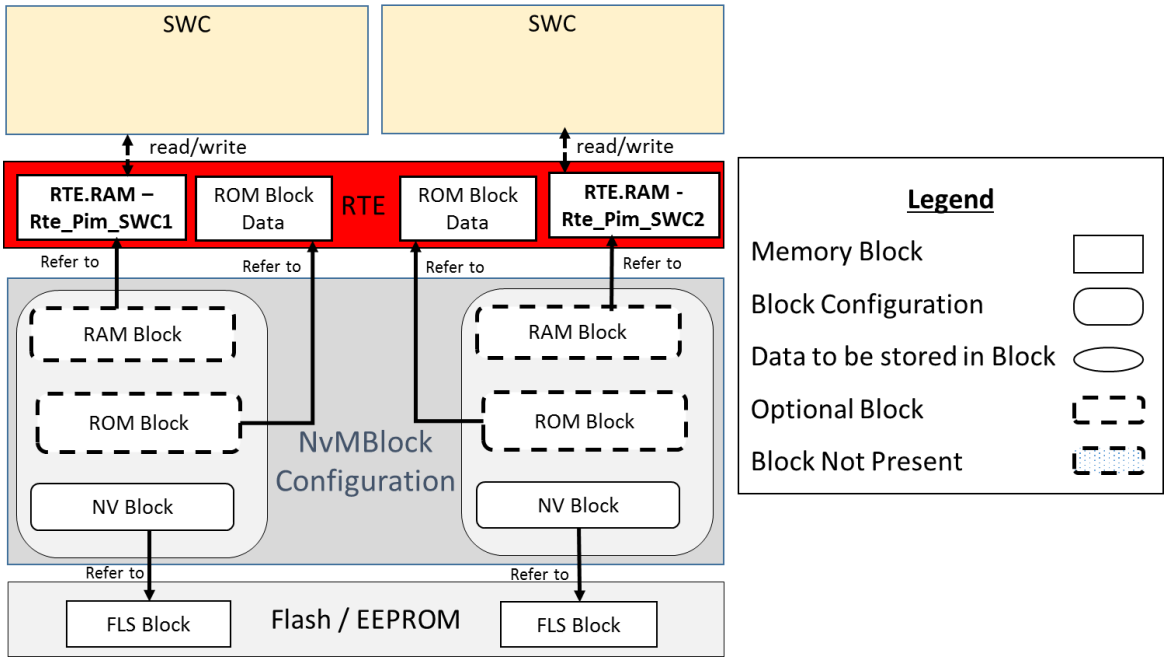


Figure 11: Overview of memory allocation for use case 2

Figure 12 shows port configuration used to access NvM interfaces by user (SWC1 / SWC2), here NvM is configured as ServiceSwComponentType.

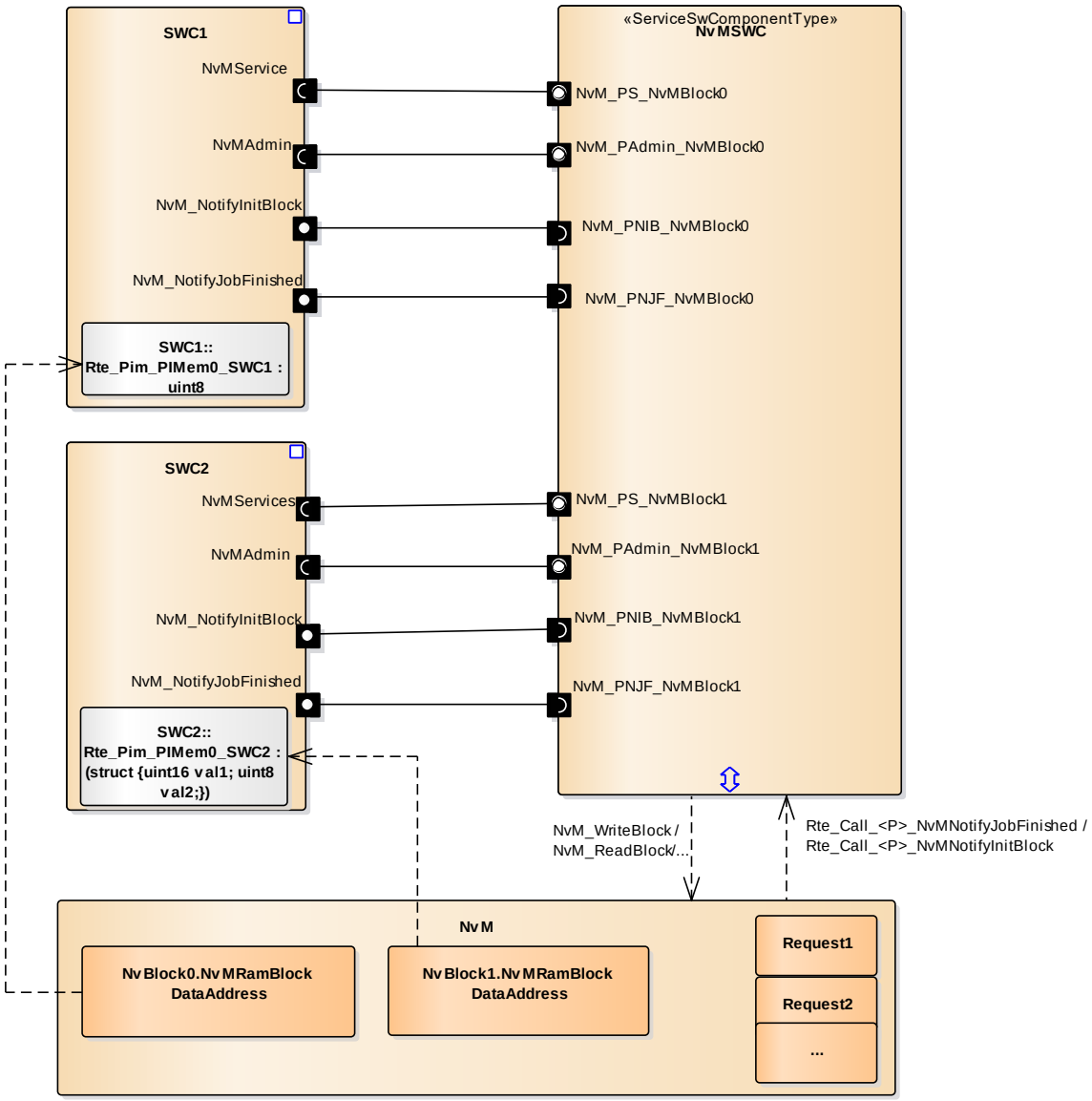


Figure 12: Port configuration for use case 2

Refer section 5.1.1 for details of opertation for ports NvMService, NvMAdmin, NvM_NotifyInitBlock and NvM_NotifyJobFinished.

Figure 13 shows how application can access NvM in this scenario:

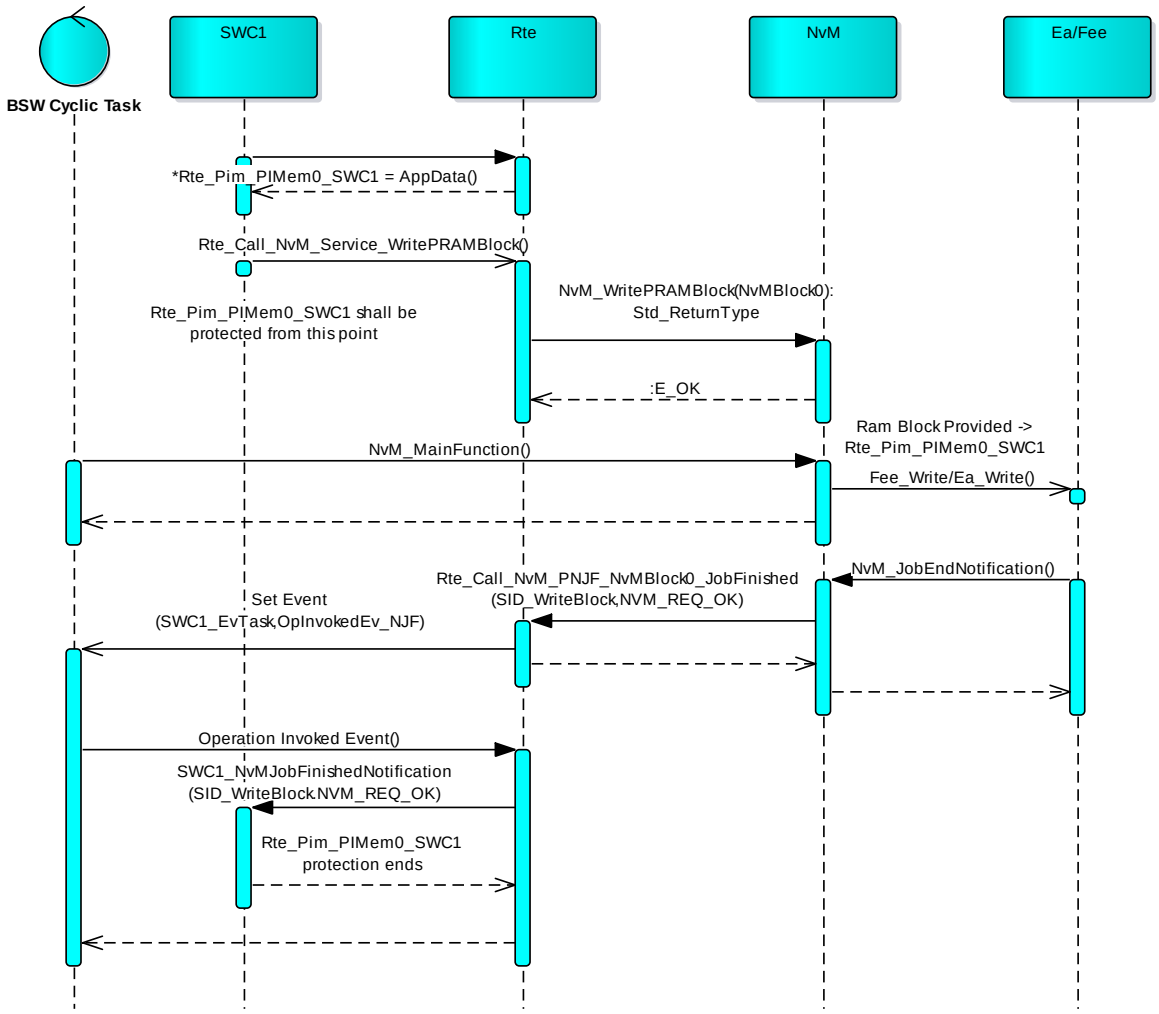


Figure 13: Sequence diagram of NvM access for use case 2

- **Applicability:**

  RTE does not ensure data consistency for the access to Per Instance Memory. An application is responsible for consistency of accessed data. For Data consistency, application should not access NV Data after call to *NvM_ReadBlock* / *NvM_WriteBlock* until it has been notified - via the JobFinished callback - that the job was successful.

  The PerInstanceMemory is allocated as RAM block in the RTE and can only be accessed by that SW-C. Thus, same data cannot be shared between two or more SW-C's. The "Access to NVRAM blocks" should not be used in multi core environments in this use case.

- **Configuration:**
  Refer below complete configuration prepared for this use case
  Configuration file –
  ConfigurationFiles/
  NvDataHandling_ServiceSwComponent_PerInstanceMemory/.arxml*

Configuration for service ports NvMService, NvMAdmin, NvM_NotifyInitBlock and NvM_NotifyJobFinished is the same as provided in section 5.1.1.

Refer below table for configuration for Per Instance Memory:

For SWC1:

Per Instance Memory	PIMem0_SWC1
Type-Definition	uint8
Owner	SWC1

For SWC2:

Per Instance Memory	PIMem0_SWC2
Type-Definition	struct {uint16 val1; uint8 val2;}
Owner	SWC2

5.3 Case 3: Application SW-C accessing NVRAM block using an NvBlockSwComponentType

In this scenario, an NvBlockSwComponent in RTE is used to create RAM Blocks as part of its NvBlockDescriptor which can be written/read partially or completely by a single or multiple SW-C's using a NV-Data Interface. These RAM blocks are accessed using mirrors in the NvM. Optionally ROM Blocks can be created as part of NvBlockDescriptor.

Along with this, the Client-Server Interfaces are used for the NvM services and notifications.

NvM Block Descriptor is configured to have explicit synchronization for this use case (*NvMBlockUseSyncMechanism* parameter set to true).

Moreover, optionally if configured, Rte can trigger data writing internally to NvM using the NvBlockSwComponent so that the Application does not have to explicitly utilize the Client Server Interface for writing.

Note: - In case of *dirtyFlag*=TRUE, NvM_WriteBlock/NvM_SetRamBlockStatus are not requested via Client Server Interfaces. In case of *dirtyFlag*=FALSE, all services are requested via Client Server Interfaces

As explained above in section 4.1.3.2, whenever NvM is triggered to write or read data from the NVRAM, it fetches/updates the RAM block via callbacks (*Rte_GetMirror/Rte_SetMirror*) which are implemented in RTE.

Additionally, if the User has a need to provide temporary RAM blocks even though mirrors are configured in the system, *NvM_ReadBlock/NvM_WriteBlock* API are used

to provide the temporary RAM block address and in that case NvM uses the RAM block argument instead of the mirror. Such API's have precedence over *NvM_ReadPRAMBlock/NvM_WritePRAMBlock*.

Note: - Implicit synchronization is not recommendable while using NvBlockSwComponentType.

Figure 14 provides an overview of memory allocation in Application and NvM as per this use case. For details about interaction between these modules refer figure 16 to 23.

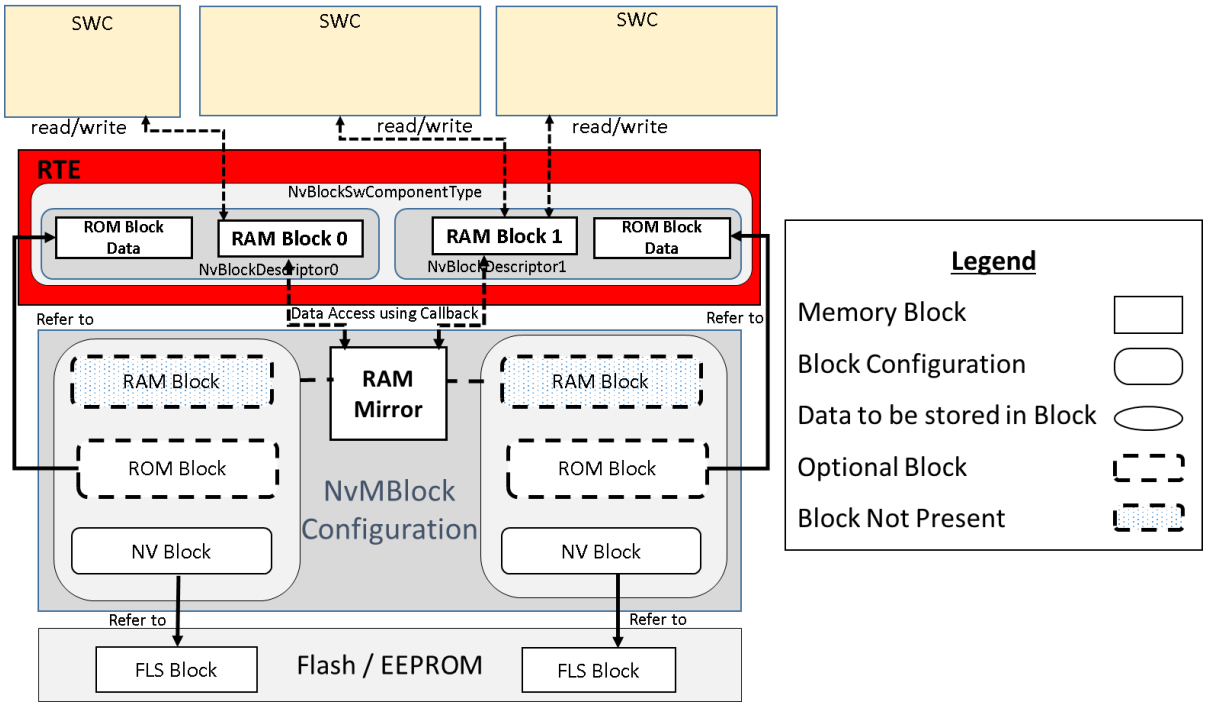


Figure 14: Overview of memory allocation for use case 3

Figure 15 shows port configuration used to access NvM interfaces by user (SWC1 / SWC2 /SWC3), here NvM is configured as NvBlockSwComponent.

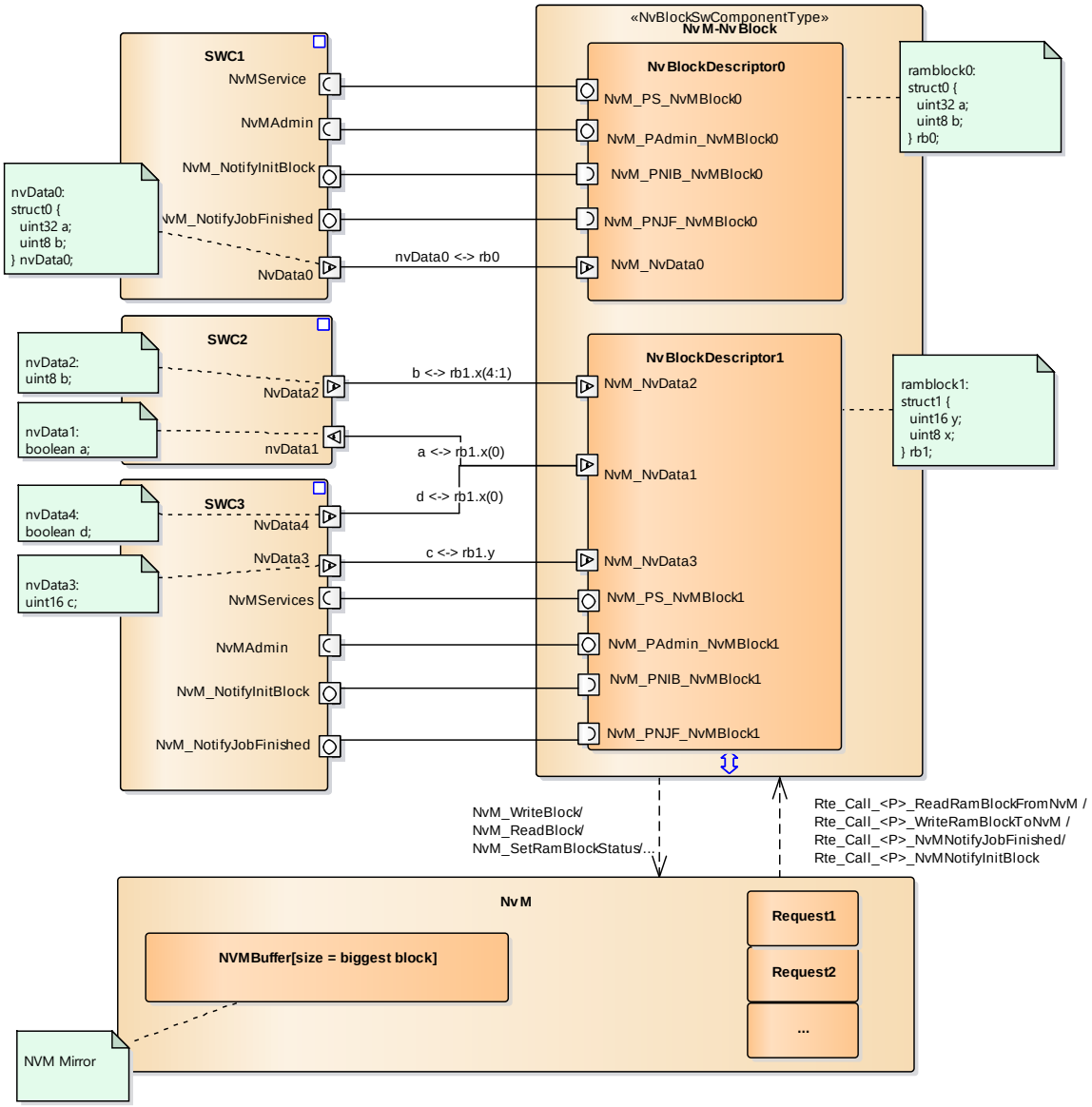


Figure 15: Port configuration for use case 3

Refer section 5.1.1 for details with respect to operations provided by the Client Server Interfaces for ports NvMService, NvMAdmin, NvM_NotifyInitBlock and NvM_NotifyJobFinished.

Note:- Mapping between NVRAM BlockId of NvM and NvBlockDescriptor is done using configuration parameters *RteNvmBlockRef* and *RteSwNvBlockDescriptorRef* of *RteNvRamAllocation* container in RTE.

NV Data mapping considered in the above example is described in below table.

nvData Element	RAM Block Element	User	NvBlockDescriptor	Description
nvData0	rb0	SWC1	NvBlockDescriptor0	nvData0 is completely mapped to rb0
nvData2.b	rb1.x(4:1)	SWC2	NvBlockDescriptor1	Bits 0 to 3 of variable b are mapped to Bits 4 to 1 of element x of rb1.
nvData1.a	rb1.x(0)	SWC2	NvBlockDescriptor1	Boolean variable a is mapped to Bit 0 of element x of rb1.
nvData4.d	rb1.x(0)	SWC3	NvBlockDescriptor1	Boolean variable d is mapped to Bit 0 of element x of rb1.
nvData3.c	rb1.y	SWC3	NvBlockDescriptor1	Variable c is completely mapped element y of rb1.

- **Applicability:**
  This use case gives advantage over other use cases as one RAM block can be shared between multiple SW-C's or BSW, completely or partially and thus reduce resource overhead. Also Data mapping allows accessing NV data without need of additional buffers.

  Different writing strategies which define the frequency of writing NV data to NV memory are possible and RTE is responsible to handle this writing operations.

  It is the responsibility of the RTE to ensure data consistency while NV Data is accessed.

- **Configuration:**

Configuration for NvData ports is shown below:

User	User Data	User Port	AccessType	NvBlockDescriptor	Intermediate mapped NvData	Mapping to Ram Block
SWC1	NvD_str0_0	PR_NvIf_strSig	Write/Read	NvBD_0	NvD_str0_0	NVBD_0.RAMBlock = NvD_str0_0
SWC2	NvD_flag0_0	R_NvIf_flagSig	Read	NvBD_1	NvD_8bit_0.b(0)	NVBD_1.RAMBlock.x = NvD_8bit_0
SWC2	NvD_4bit_0	PR_NvIf_4bitSig	Write/Read	NvBD_1	NvD_8bit_0.b(4:1)	NVBD_1.RAMBlock.x = NvD_8bit_0
SWC3	NvD_16bit_0	PR_NvIf_16bitSig	Write/Read	NvBD_1	NvD_16bit_0	NVBD_1.RAMBlock.y = NvD_16bit_0
SWC3	NvD_flag0_0	P_NvIf_flagSig	Write	NvBD_1	NvD_8bit_0.b(0)	NVBD_1.RAMBlock.x = NvD_8bit_0

Table 1

Data	Type
NvD_str0_0	struct {uint32 a; uint8 b;}
NvD_flag0_0	boolean
NvD_4bit_0	uint8
NvD_16bit_0	uin16
NvD_8bit_0	uint8
NVBD_0.RAMBlock	struct {uint32 a; uint8 b;}
NVBD_1.RAMBlock	struct {uint16 x; uint8 y;}

Mapping between NvBlockDescriptor and NvM block Id is done using *RteNvRamAllocation* container in RTE.

RteNvmBlockRef	RteSwNvBlockDescriptorRef
/NvM0/NvMBlockDescriptor0	/NvM_Service/Component/NvM/NVBD_0
/NvM0/NvMBlockDescriptor1	/NvM_Service/Component/NvM/NVBD_1

This scenario is further categorized depending on the type of sender-receiver communication used by the Application software to access the RAM block and the respective writing strategy.

5.3.1 Case 3a: Using Rte Explicit S/R Communication

In this scenario, Rte Explicit API's i.e. *Rte_Write/Rte_Read/Rte_DRead* are used to update or read the RAM block or an element of a RAM block. Rte Explicit API's normally perform the writing or reading operation before the Rte request returns.

The Rte Explicit API simply modify the RAM block in the most basic use-case. The scope of the Rte Explicit request can be extended using the configuration of the NvBlockDescriptor the RAM block of which is being updated.

- **Configuration:**
  Refer below complete configuration prepared for this use case
  Configuration file –
  ConfigurationFiles/
  NvDataHandling_NvBlockSwComponent_RteExplicitCommunication/.arxml*

There are four types of scenarios possible with respect to the writing strategy between RTE and NvM:

5.3.1.1 No Dirty Flag Support

RTE does not implicitly trigger NvM for the writing operation. The user SW-C has to initiate the request using the Client-Server port Interface.

This use-case is applicable when the data-writing behavior needs to be controlled by the Application SW-C itself.

Data consistency is ensured by RTE.

Figure 16 shows how application can access NvM in this scenario:

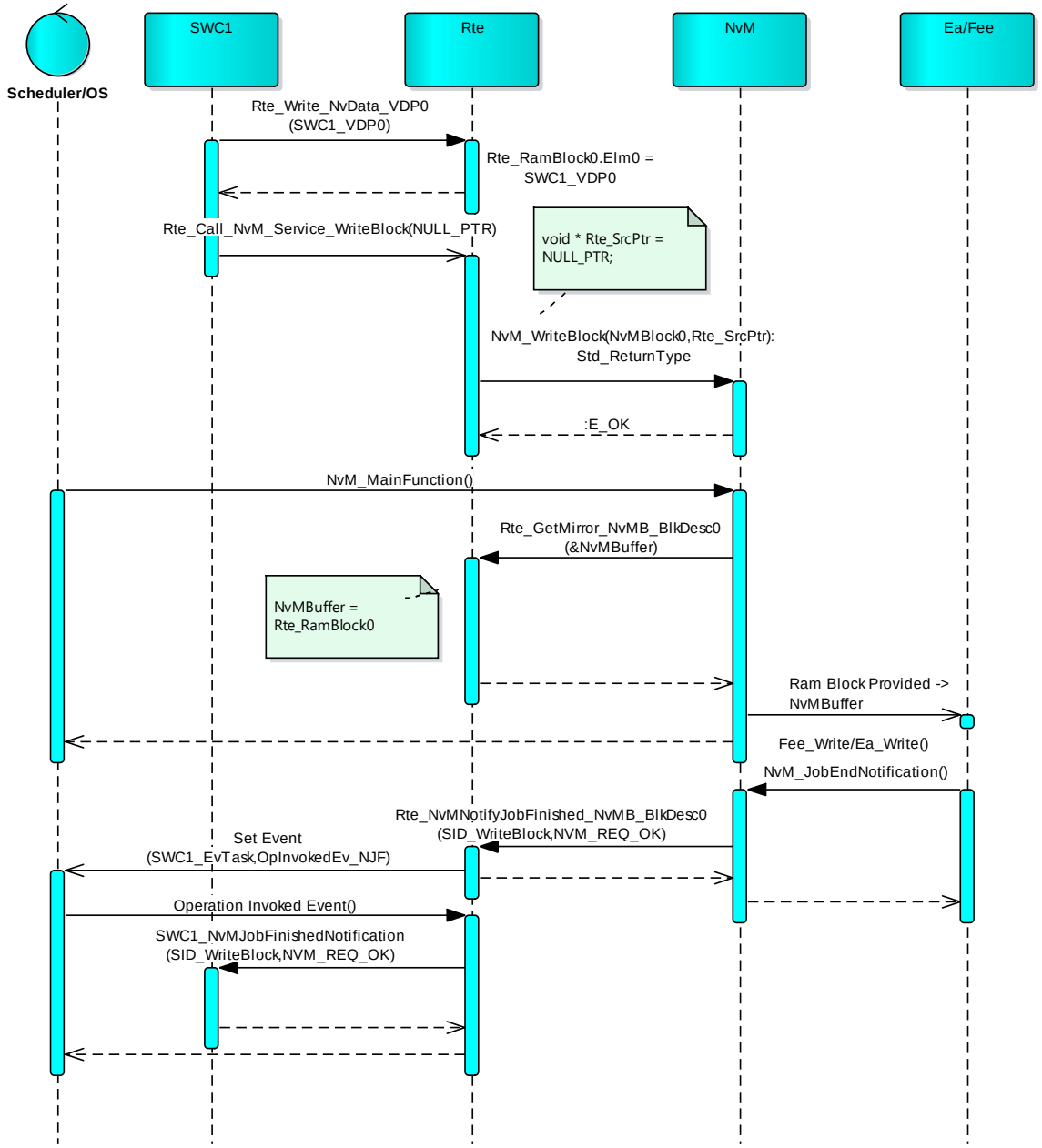


Figure 16: Sequence diagram of NvM access for use case 3a – No dirty flag support

- **Configuration:**

Port configuration is the same as mentioned in Table 1. In this use case, *supportDirtyFlag* parameter for NvBD_0 NvBlockDescriptor is configured as False.

As DirtyFlag support is disabled, *NvBlockNeeds* configuration is not applicable.

5.3.1.2 With Dirty Flag Support

5.3.1.2.1 Storing Cyclically

For a RAM block for which *dirtyFlagSupport* is set to true and *NvBlockNeeds*.*storeCyclic* is set to true, a timing event handled by the RTE periodically triggers writing of the RAM block into NVRAM by the NvM. The periodicity of the timing event is configured as part of NvBlockDescriptor.

Figure 17 shows how application can access NvM in this scenario:

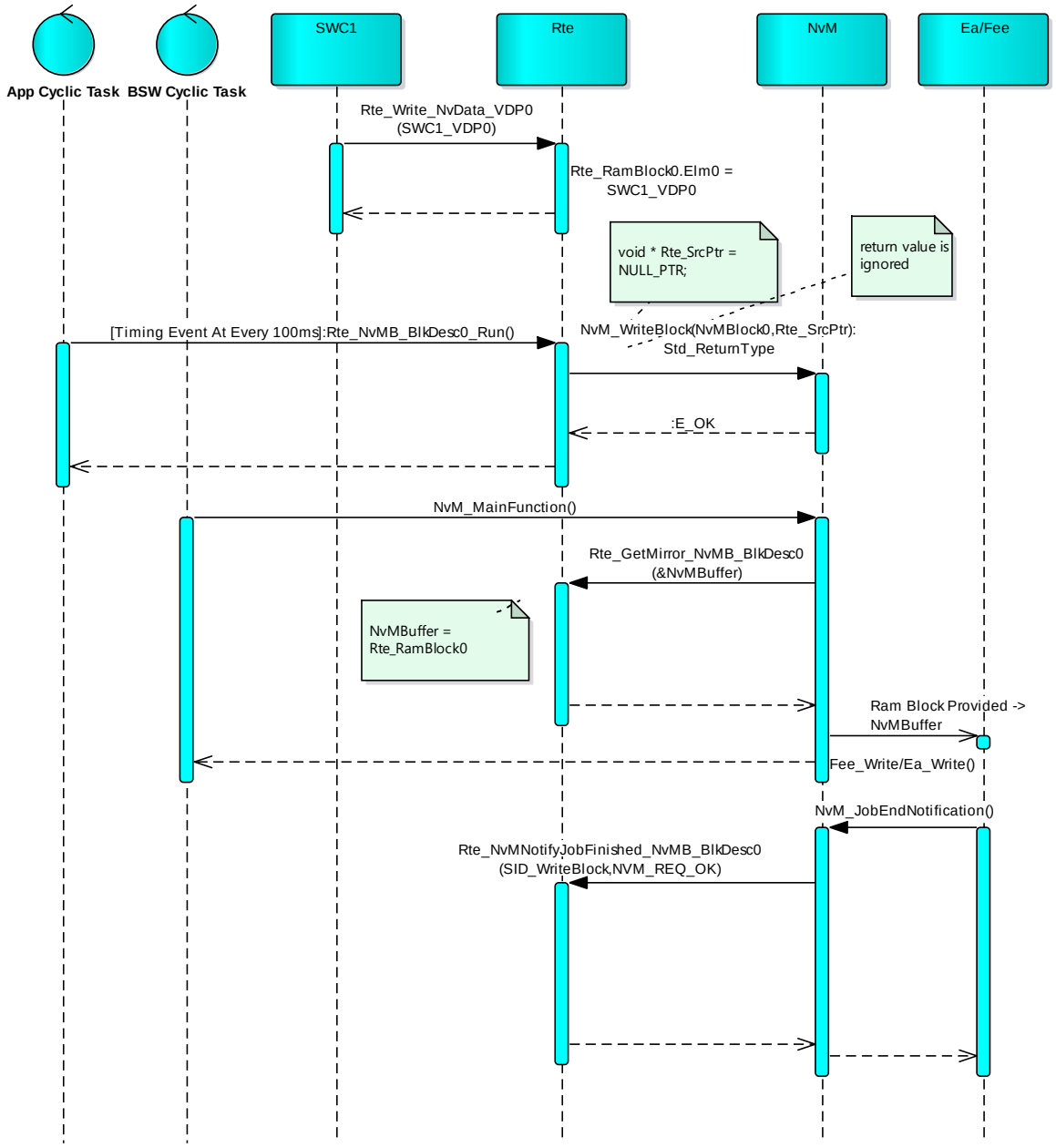


Figure 17: Sequence diagram of NvM access for use case 3a – Store Cyclic

- **Configuration:**
  Port configuration is the same as mentioned in Table 1. In this use case, *supportDirtyFlag* parameter for NvBD_1 NvBlockDescriptor is configured as True. And following *NvBlockNeeds* configuration is provided for store cyclic case.

NvBlockNeeds Parameter	Value
storeAtShutdown	FALSE
storeCyclic	TRUE
storeImmediate	FALSE
writingFrequency	0.1

5.3.1.2.2 Storing at Shutdown

On the update of a RAM block for which *dirtyFlagSupport* is set to true and *NvBlockNeeds.storeAtShutdown* is set to true, the RTE invokes the *NvM_SetRamBlockStatus* function of the NvM module with the BlockChanged parameter set to true. BswM is responsible for writing this changed blocks during shutdown.

A DataReceivedEvent needs to be configured for the NvBlockNeeds of this NvBlockDescriptor. If this DataReceivedEvent is not mapped to any task, then RTE calls the *NvM_SetRamBlockStatus* API from inside the *Rte_Write* API context.

Figure 18 shows how application can access NvM in this scenario:

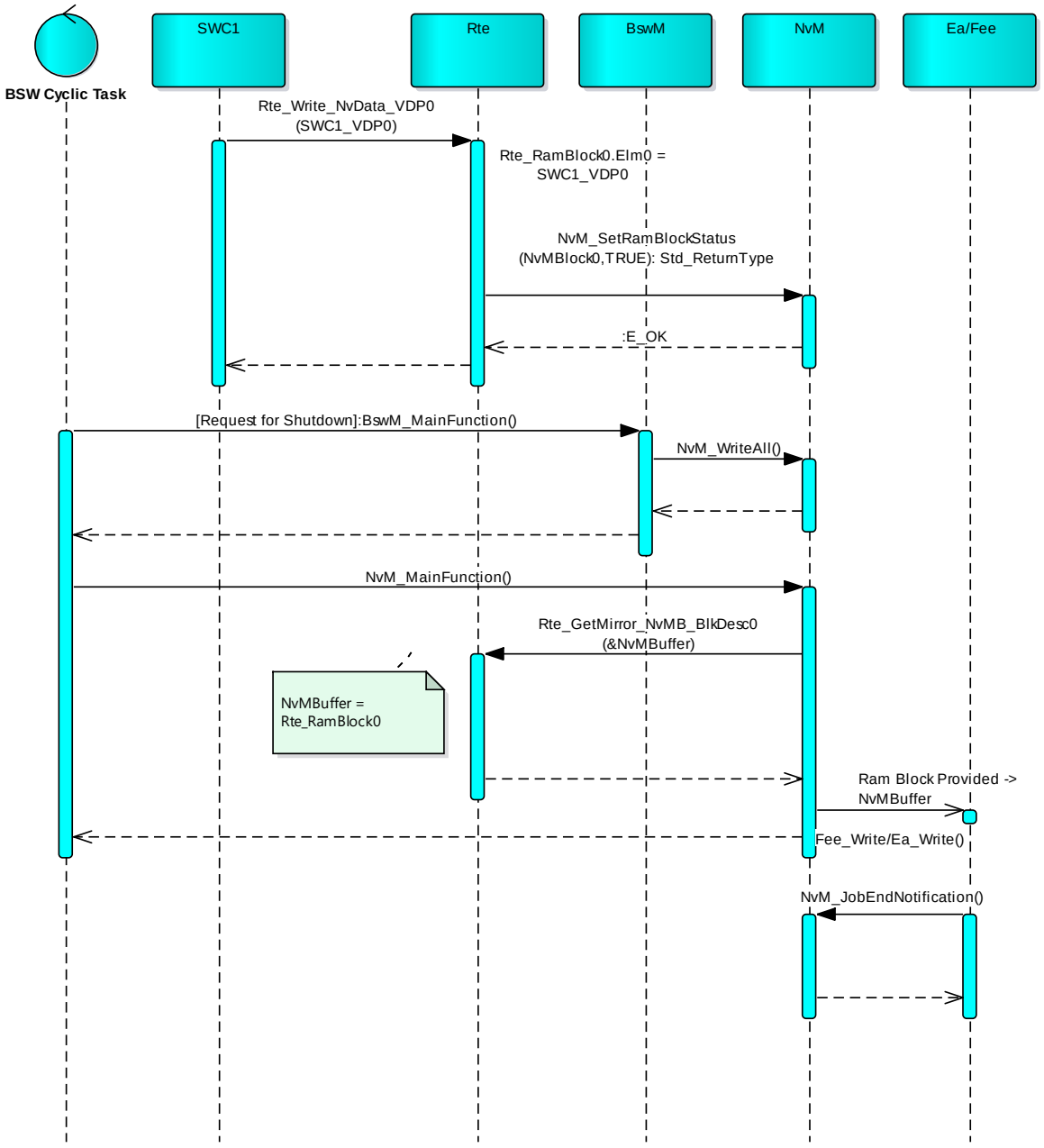


Figure 18: Sequence diagram of NvM access for use case 3a – Store at shutdown

- **Configuration:**
  Port configuration is the same as mentioned in Table 1. In this use case, *supportDirtyFlag* parameter for NvBD_1 NvBlockDescriptor is configured as True. And following NvBlockNeeds configuration is provided.

NvBlockNeeds Parameter	Value
storeAtShutdown	TRUE
storeCyclic	FALSE
storeImmediate	FALSE
writingFrequency	-

5.3.1.2.3 Storing Immediately

On the update of a RAM block for which *dirtyFlagSupport* is set to true and *NvBlockNeeds.storeImmediate* is set to true, the RTE invokes the NvM_WriteBlock function of the NvM module with the BlockId mapped using RteNvRamAllocation.

There are two options through which this can be achieved:

1. Using Task Activation, a DataReceivedEvent is configured for the NvBlockNeeds of this NvBlockDescriptor. If this DataReceivedEvent is mapped to a task, then RTE activates the task from inside the Rte_Write API context. The runnable mapped to the DataReceivedEvent is implemented by RTE to call the *NvM_WriteBlock* API.

2. From Request context, a DataReceivedEvent is configured for the NvBlockNeeds of this NvBlockDescriptor. If this DataReceivedEvent is not mapped to any task, then RTE calls the *NvM_WriteBlock* API from inside the Rte_Write API context.

Figure 19 shows how application can access NvM in this scenario:

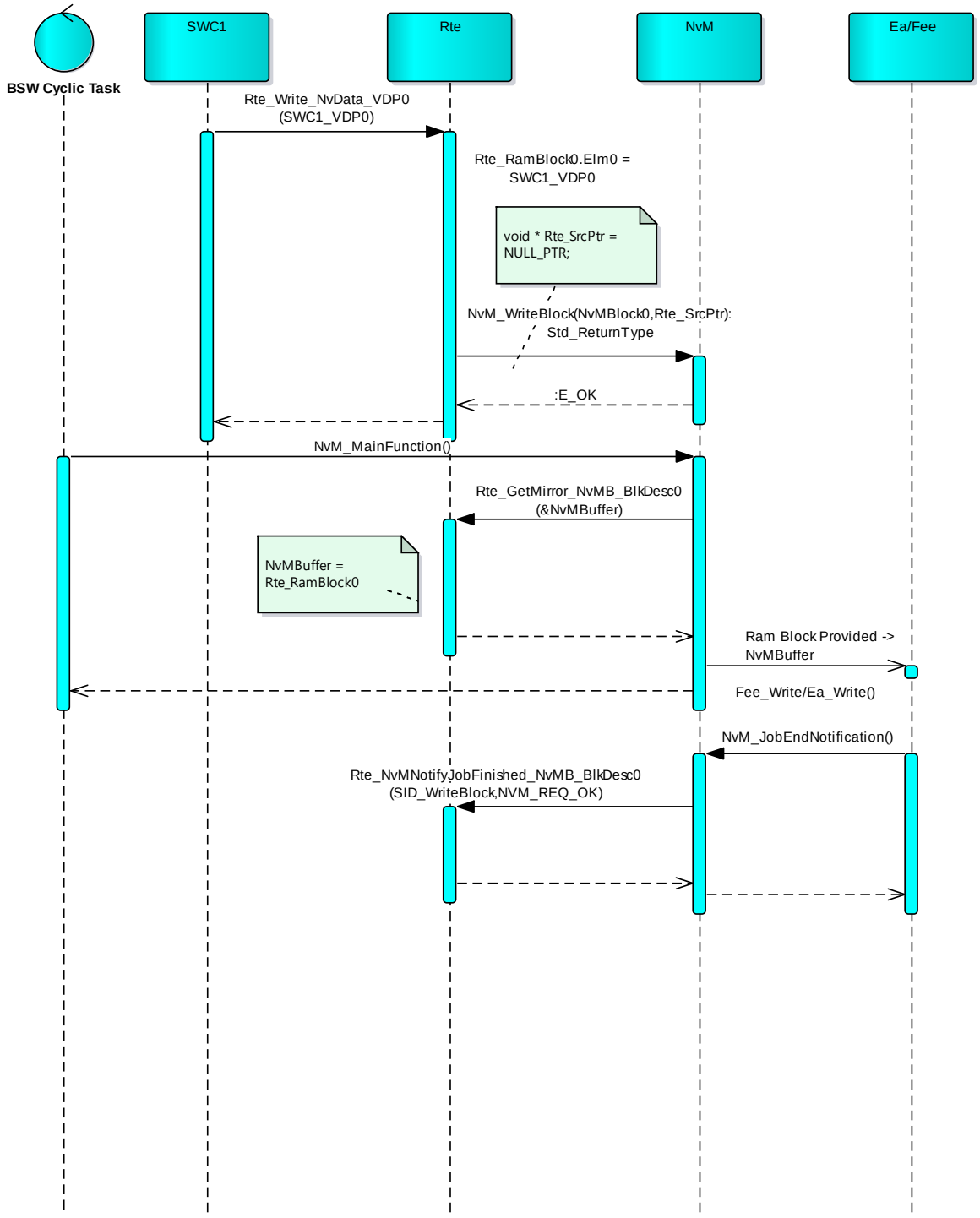


Figure 19: Sequence diagram of NvM access for use case 3a – Store Immediate

- **Configuration:**

  Port configuration is the same as mentioned in Table 1. In this use case, supportDirtyFlag parameter for NvBD_1 NvBlockDescriptor is configured as True. And following NvBlockNeeds configuration is provided.

NvBlockNeeds Parameter	Value
storeAtShutdown	FALSE
storeCyclic	FALSE
storeImmediate	TRUE
writingFrequency	-

5.3.2 Case 3b: Using Rte Implicit S/R Communication

In this scenario, Rte Implicit API's i.e. *Rte_IWrite/Rte_IWriteRef/Rte_IRead* are used to update or read the RAM block or an element of a RAM block. Rte Implicit API's normally buffer the data to be written intermediately until the runnable from which the Rte implicit write request is called, terminates. After the termination of the runnable, the global RAM block maintained by the RTE is updated with the last written value. In case of implicit read request, the data is fetched from the RAM block before the runnable entity which has read access to the data is activated.

The Rte Implicit API simply modify the RAM block in the most basic use-case. The scope of the Rte Implicit access can be extended using the configuration of the NvBlockDescriptor the RAM block of which is being updated.

- **Configuration:**
  Refer below complete configuration prepared for this use case
  Configuration file –
  ConfigurationFiles/
  NvDataHandling_NvBlockSwComponent_RteImplicitCommunication/.arxml*

There are four types of scenarios possible with respect to the writing strategy which RTE can trigger the writing operation of NvM:

5.3.2.1 No Dirty Flag Support

RTE does not implicitly trigger NvM for the writing operation. The user SW-C has to initiate the request using the Client-Server port Interface.

This use-case is applicable when the data-writing behavior needs to be controlled by the Application SW-C itself.

Figure 20 shows how application can access NvM in this scenario:

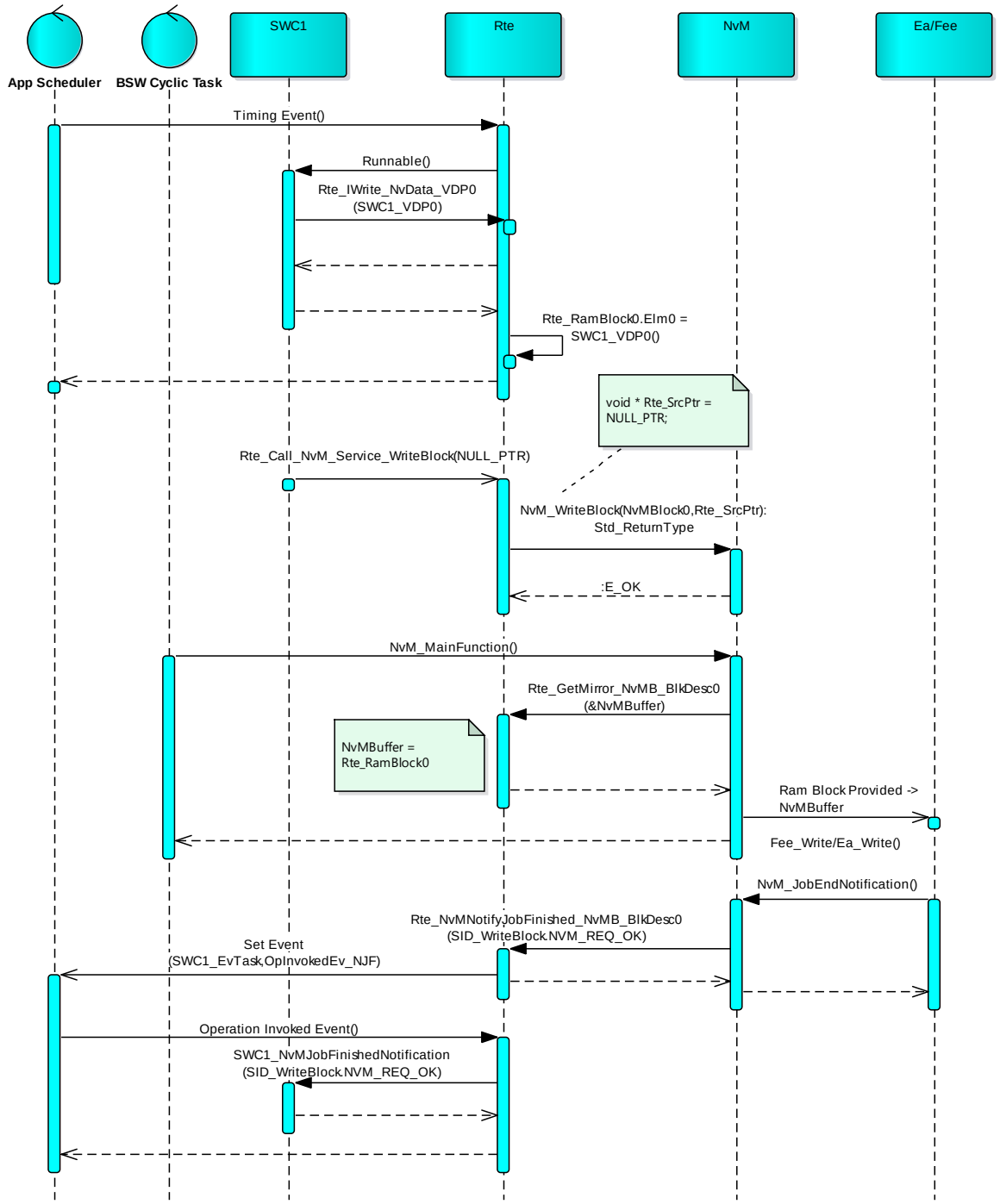


Figure 20: Sequence diagram of NvM access for use case 3b – No dirty flag support

- **Configuration:**

  Port configuration is the same as mentioned in Table 1. In this use case, supportDirtyFlag parameter for NvBD_0 NvBlockDescriptor is configured as False. As DirtyFlag support is disabled, NvBlockNeeds configuration is not applicable.

5.3.2.2 With Dirty Flag Support

5.3.2.2.1 Storing Cyclically

For a RAM block for which *dirtyFlagSupport* is set to true and *NvBlockNeeds.storeCyclic* is set to true, a timing event handled by the RTE

periodically triggers writing of the RAM block into NVRAM by the NvM. The periodicity of the timing event is configured as part of NvBlockDescriptor.

Figure 21 shows how application can access NvM in this scenario:

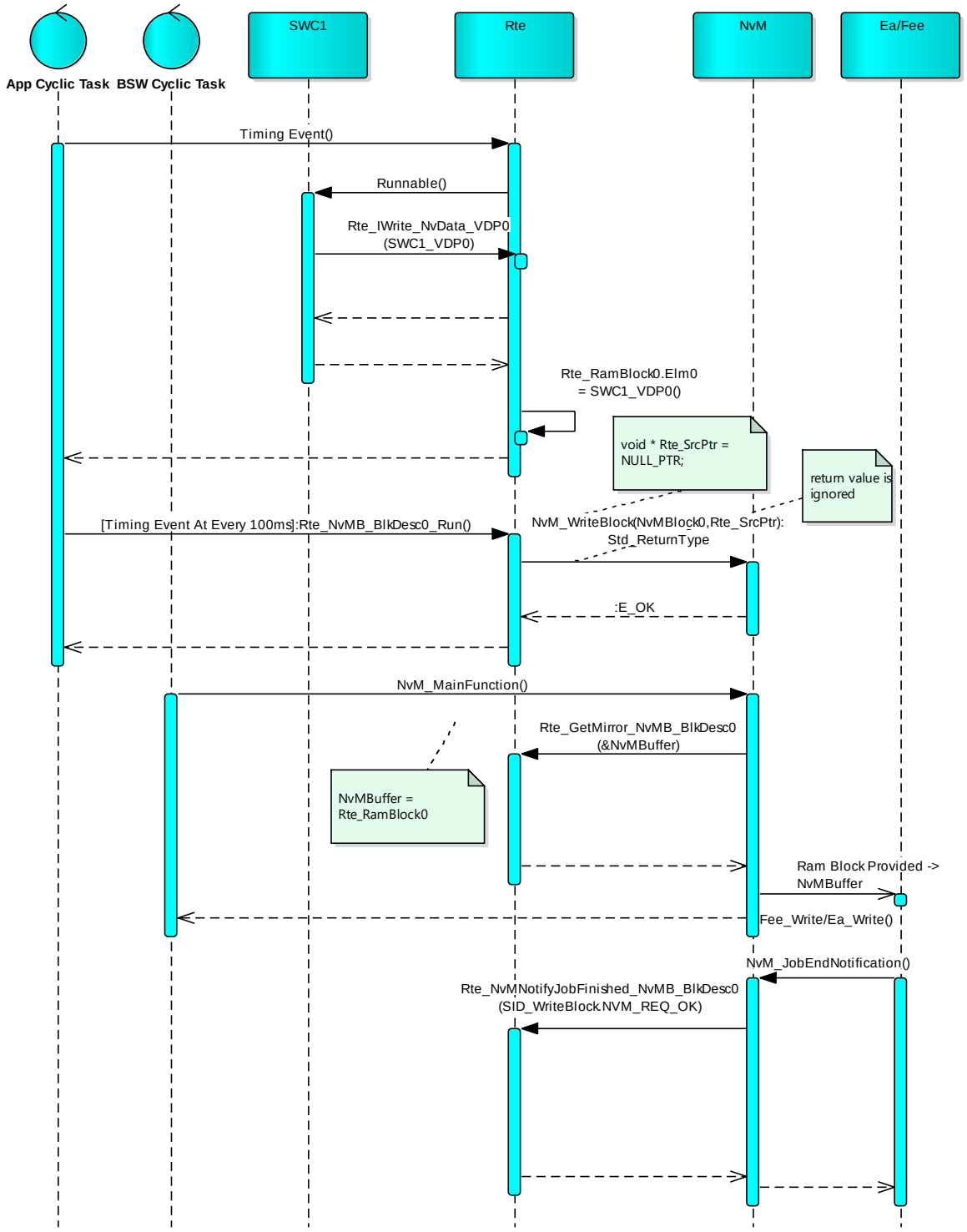


Figure 21: Sequence diagram of NvM access for use case 3b – Store cyclic

- **Configuration:**

  Port configuration is the same as mentioned in Table 1. In this use case, supportDirtyFlag parameter for NvBD_1 NvBlockDescriptor is configured as True. And following NvBlockNeeds configuration is provided for store cyclic case.

NvBlockNeeds Parameter	Value
storeAtShutdown	FALSE
storeCyclic	TRUE
storeImmediate	FALSE
writingFrequency	0.1

5.3.2.2.2 Storing at Shutdown

On the update of a RAM block for which *dirtyFlagSupport* is set to true and *NvBlockNeeds.storeAtShutdown* is set to true, the RTE invokes the *NvM_SetRamBlockStatus* function of the NvM module with the BlockChanged parameter set to true. A DataReceivedEvent is configured for the NvBlockNeeds of this NvBlockDescriptor. If this DataReceivedEvent is not mapped to any task, then RTE calls the *NvM_SetRamBlockStatus* API from inside the context of the task mapped to the runnable with the write access. This is done after the runnable with the write access terminates.

Figure 22 shows how application can access NvM in this scenario:

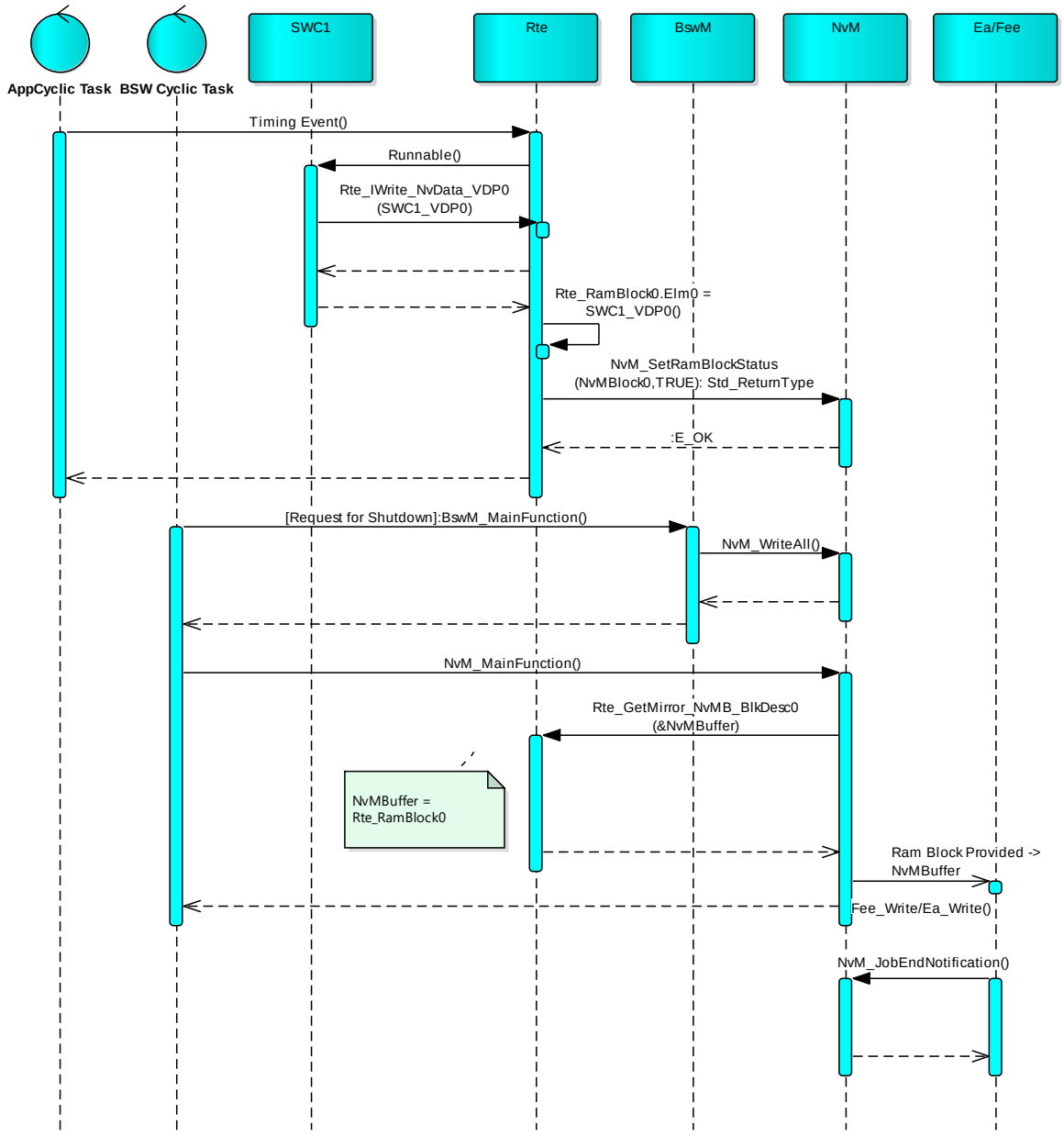


Figure 22: Sequence diagram of NvM access for use case 3b – Store at shutdown

- **Configuration:**
  Port configuration is the same as mentioned in Table 1. In this use case, supportDirtyFlag parameter for NvBD_1 NvBlockDescriptor is configured as True. And following NvBlockNeeds configuration is provided.

NvBlockNeeds Parameter	Value
storeAtShutdown	TRUE
storeCyclic	FALSE
storeImmediate	FALSE
writingFrequency	-

5.3.2.2.3 Storing Immediately

On the update of a RAM block for which *dirtyFlagSupport* is set to true and *NvBlockNeeds.storeImmediate* is set to true, the RTE invokes the *NvM_WriteBlock* function of the NvM module with the BlockId mapped using RteNvRamAllocation. There are two options through which this can be achieved:

1. Using Task Activation, a DataReceivedEvent is configured for the NvBlockNeeds of this NvBlockDescriptor. If this DataReceivedEvent is mapped to a task, then RTE activates the task from the context of the task mapped to the runnable with the write access. This is done after the runnable with the write access terminates. The runnable mapped to the DataReceivedEvent is implemented by RTE to call the *NvM_WriteBlock* API.

2. From Request context, a DataReceivedEvent is configured for the NvBlockNeeds of this NvBlockDescriptor. If this DataReceivedEvent is not mapped to any task, then RTE calls the *NvM_WriteBlock* API from inside the context of the task mapped to the runnable with the write access. This is done after the runnable with the write access terminates.

Figure 23 shows how application can access NvM in this scenario:

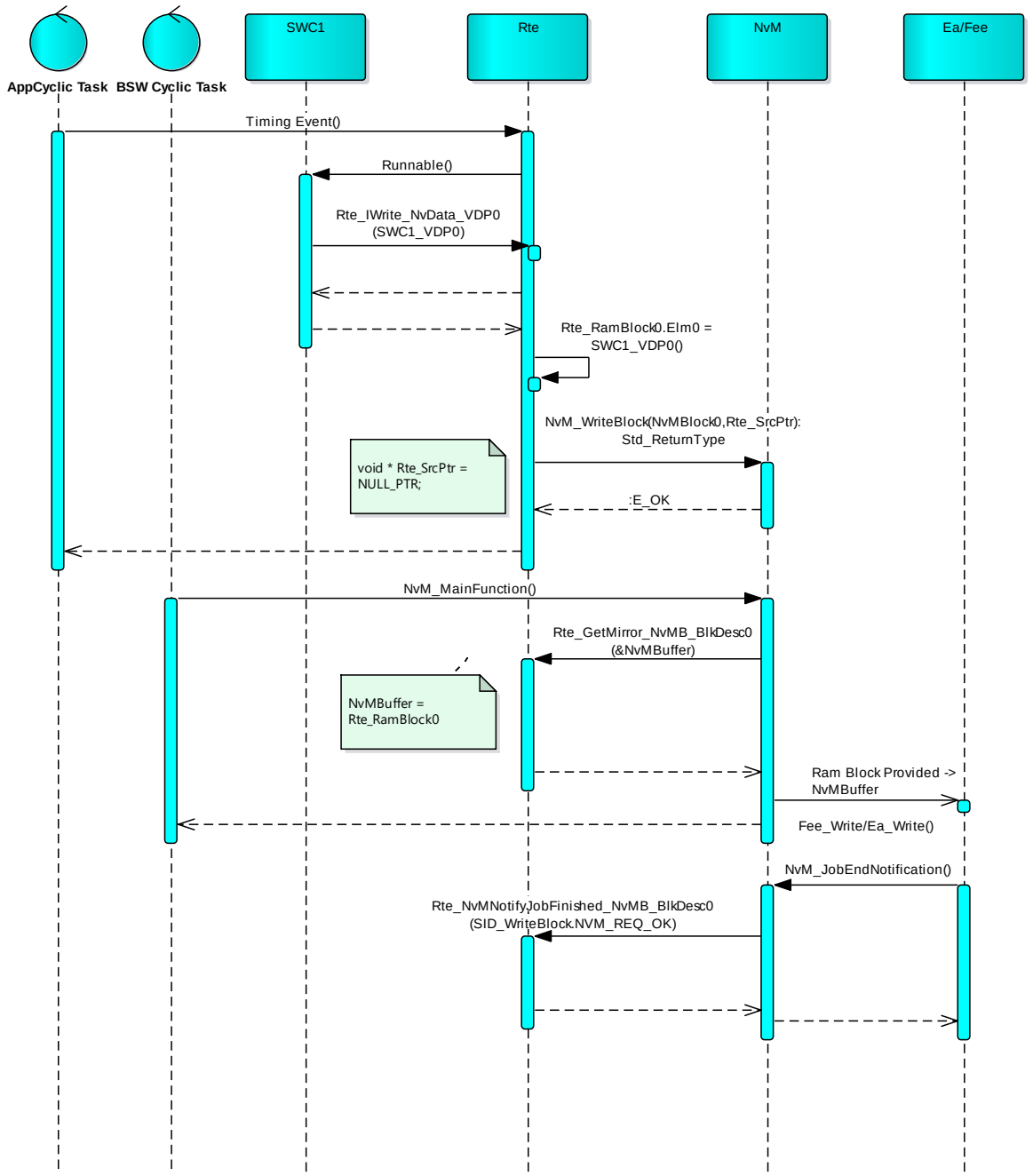


Figure 23: Sequence diagram of NvM access for use case 3b – Store Immediately

- **Configuration:**
  Port configuration is the same as mentioned in Table 1. In this use case, supportDirtyFlag parameter for NvBD_1 NvBlockDescriptor is configured as True. And following NvBlockNeeds configuration is provided.

NvBlockNeeds Parameter	Value
storeAtShutdown	FALSE
storeCyclic	FALSE
storeImmediate	TRUE

6 Appendix

Following table provides an overview of usage of RAM / ROM / NVM Mirror blocks for different use cases described above.

	RAM Block	ROM Block	NVM Mirror
Use case 1a	NA	Mapped to RTE ROM data	NA
Use case 1b	NA	Mapped to RTE ROM data	Present in NVM
Use case 2	Mapped to RTE PIM	Mapped to RTE ROM data	NA
Use case 3	Present in RTE	Mapped to RTE ROM data	Present in NVM