VECTOR >

MICROSAR EA

Technical Reference

Version 4.00.00

Authors	virdu, virgmi, virbka, virbmz
Status	Released

Document Information

History

Author	Date	Version	Remarks
Peter Paulus	2007-02-27	1.0	Creation of the document (as user manual).
Peter Paulus	2007-07-09	1.1	Revision due to development of EA SWS architecture version 1.1.0.
Peter Paulus	2007-08-07	1.2	Revision due to review.
Peter Paulus	2007-10-30	1.3	Revision due to comparison with FEE user manual. Added configuration of EA Address Type and EA Length Type.
Peter Paulus	2007-12-19	2.00.00	Changed format to a three digit number.
Peter Paulus, Heike Bischof	2008-05-09	3.00.00	Conversion to Technical Reference
Peter Paulus	2008-11-04	3.00.01	Revision due to review.
virdu	2009-04-29	3.01.00	Adaption to AUTOSAR 3 ESCAN00032794: chapter 6; Configuration tab "Published Information" was removed. ESCAN00032762: chapter 6.1.3.1; Development Error Detection and Reporting changed.
virdu	2010-11-08	3.01.01	ESCAN00043763 Chapter 4.3: Changed section to standard section START_SEC_VAR_FAST_NOINIT_UNSPECIFIED.
virdu	2013-02-20	3.02.00	Ch. 6: Configuration
virdu	2013-09-20	3.02.01	Ch: 3.5.1 Development Error Reporting updated

virdu	2013-12-13	3.03.00	partition concept added:
			  Ch. 3.1 Features updated
			  Ch. 3.6 Partitions added

virgmi	2015-04-16	3.04.00	Mainly feature tables were updated due to SafeBSW implementation of EA.
virbka	2021-03-18	3.04.01	Added information about block storage in chapter 2.7.
virbmz	2021-05-21	3.04.02	Added information about new memory sections in chapter 3.1.3.
virbka	2022-07-22	4.00.00	Added information about new memmap file generation in chapter 3.1.3.

Reference Documents

No.	Source	Title	Version
[1]	AUTOSAR	Specification of EEPROM Abstraction	V2.0.0
[2]	AUTOSAR	Specification of Development Error Tracer	V3.2.0
[3]	AUTOSAR	List of Basic Software Modules	V1.6.0
[4]	Vector Informatik	MICROSAR MemMap Technical Reference	See delivery

Caution

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

Contents

Illustrations

Tables

1 Introduction

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

Latest Analyzed AUTOSAR Release:	R19-11	
AUTOSAR Schema Compatibility:	4.0.3	
Supported Configuration Variants:	pre-compile	
Vendor ID:	EA_VENDOR_ID	30 decimal (= Vector-Informatik, according to HIS)
Module ID:	EA_MODULE_ID	40 decimal (according to ref. [3])

* For the precise AUTOSAR Release 4.x please see the release specific documentation.

The EA provides access to EEPROM for reading and storing data persistently.

The EA module operates on blocks provided by the NVRAM manager. Additional to NVM blocks the EA can handle/configure user blocks, e.g. for a boot loader application.

Further on, the module depends on some other modules, like DET for error detection, the underlying EEPROM driver for hardware access and MEMIF for consistent types.

1.1 Architecture Overview

The following figure shows where the EA is located in the AUTOSAR architecture.
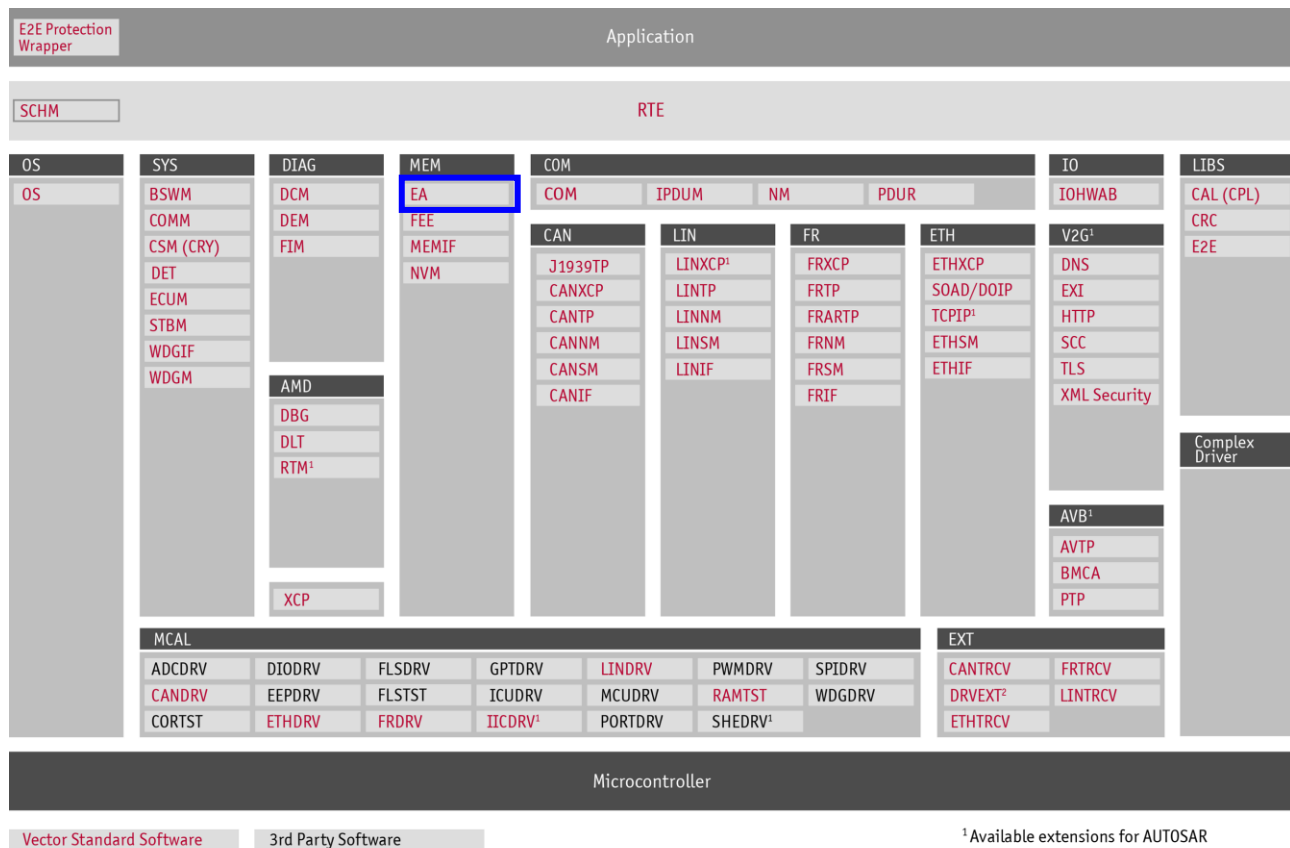
Figure 1-1   AUTOSAR architecture

The next figure shows the interfaces to adjacent modules of the EA. These interfaces are described in chapter 4.
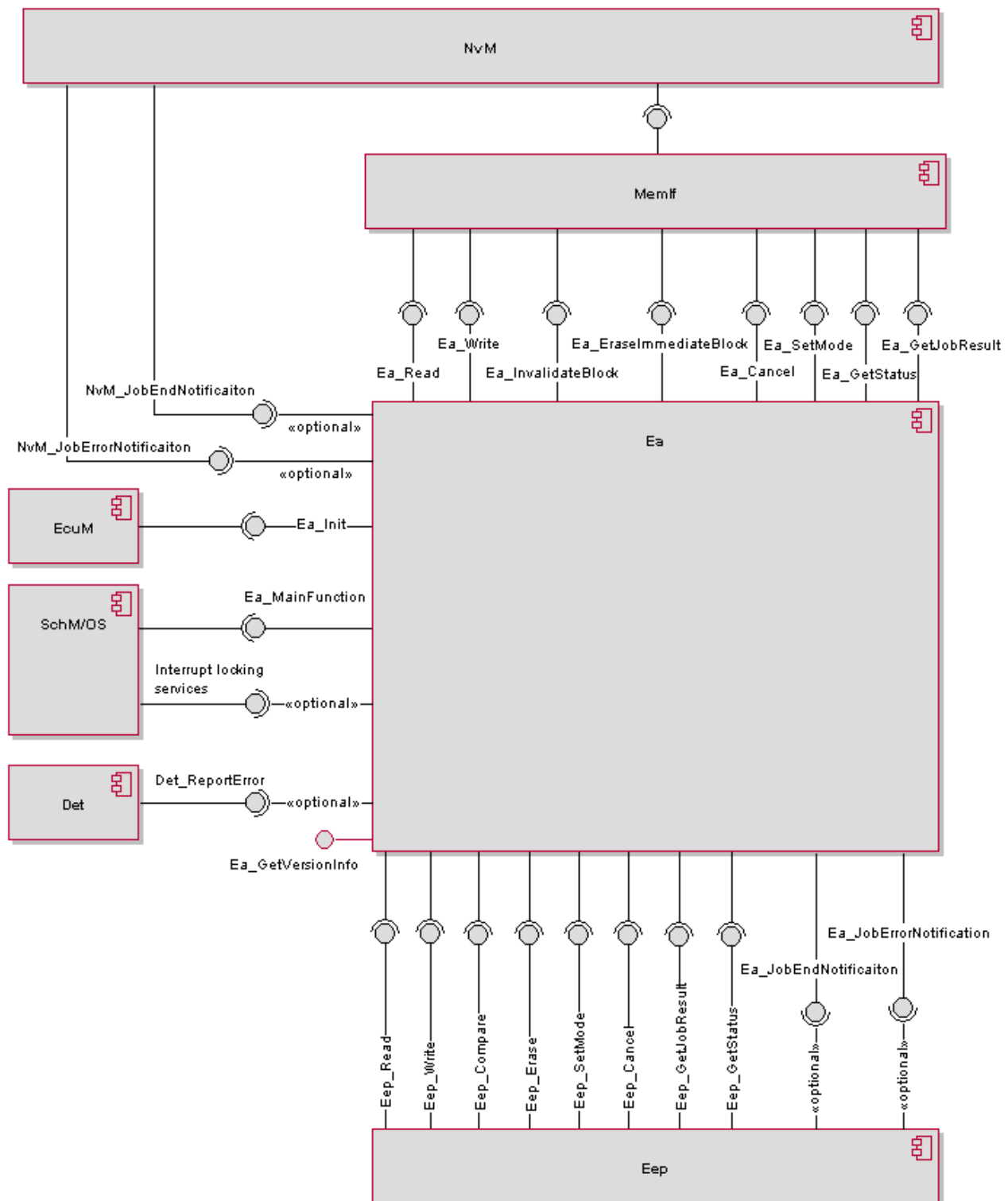
Figure 1-2   Interfaces to adjacent modules of the EA

1.2 Component Usage

Correct functionality of EA component is based on following condition related to the usage of the component:

> **!** **Caution**
> The user of EA ensures that one EA API request (e.g. Ea_Read, Ea_Write, …) is not interrupted by any second EA API request. This means that EA API requests are handled as atomic transactions by users of EA (i.e. NVM, FBL).

2 Functional Description

2.1 Features

The features listed in the following tables cover the complete functionality specified for the EA.

The AUTOSAR standard functionality is specified in [1], the corresponding features are listed in the tables

- Table 3-1 Supported AUTOSAR standard conform features
- Table 3-2 Not supported AUTOSAR standard conform features

Vector Informatik provides further EA functionality beyond the AUTOSAR standard. The corresponding features are listed in the table

- Table 3-3 Features provided beyond the AUTOSAR standard

The following features specified in [1] are supported:

| Supported AUTOSAR Standard Conform Features |
|---|
| The EA provides an interface for reading of blocks from EEPROM provided by the NVM |
| The EA provides an interface for writing blocks provided by the NVM to EEPROM |
| The EA provides an interface for erasing blocks which contain immediate data |
| The EA provides an interface for invalidating blocks provided by the NVM |
| If the underlying EEPROM device does not provide at least the configured number of erase/write cycles per physical memory cell, the EA provides a mechanism to spread the erase/write access such that the physical device is not overstressed. |
| The EA provides a header/trailer mechanism to manage each block's information, whether this block is "correct" from the point of view of the EA. |
| The EA provides development error detection to check API parameter. |
| The EEPROM driver can be polled by the EA for its current state or the EEPROM driver can provide notifications to the EA module via the callback mechanism. This configuration is only available at pre-compile time. |
| The EA can be polled by the NVM or the EA provides the result of a job to the upper layer via the callback mechanism. This configuration is only available at pre-compile time. |

Table 2-1   Supported AUTOSAR standard conform features

2.1.1 Deviations

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

Not Supported AUTOSAR Standard Conform Features
Alignment of logical blocks via VIRTUAL_PAGE_SIZE is not supported. Instead, alignment of logical blocks is provided via partition specific address alignment. Therefore, see Table 3-3
Debugging support
The maximum time the EA module's API routines shall be blocked by internal operations.
Setting mode of underlying EEPROM is not supported, because handling of set mode is not

Not Supported AUTOSAR Standard Conform Features
clearly specified and does not have any effect on internal EEPROM drivers.

Table 2-2   Not Supported AUTOSAR Standard Conform Features

2.1.2   Additions/Extensions

The following features are provided beyond the AUTOSAR standard:

Features provided beyond the AUTOSAR standard
The EA supports usage of multiple EEPROM devices.
The EA provides the configuration of partitions, in order to separate independent contents/devices from each other.
The EA provides partition specific address alignments to which all logical blocks of a partition are aligned. Address alignment is configurable separately for each partition. Address alignment must not be smaller than a partition's write alignment.
The EA provides verification of data which has been recently written to EEPROM. This feature can be configured block specifically.
The EA provides detection and correction of single bit flips in a block's management information.
The EA supports two main function triggering modes. MainFunction of EA module can either be called cyclically with a fixed cycle time or in a background task.

Table 2-3   Features provided beyond the AUTOSAR standard

2.2   Initialization

The EA is initialized and operational after the API function `Ea_Init()` has been called.

> **! Caution**
> The EA is driven asynchronously if a job has been requested to it. I.e., the jobs are finally processed by calls of the EA's MainFunction.

2.3   States

The EA can change to different states during runtime which are described in the table below.

2.3.1   Module States

Module States	Point in Time
MEMIF_UNINIT	The API service `Ea_Init()` has not been called yet. The service `Ea_GetStatus()` returns the value `MEMIF_UNINIT`.
MEMIF_IDLE	The API service `Ea_Init()` has been called and finished successfully. No job has currently been requested to the EA. The service `Ea_GetStatus()` returns the value `MEMIF_IDLE`.

`MEMIF_BUSY`	The API services `Ea_Read()`, `Ea_Write()`, `Ea_InvalidateBlock()` or `Ea_EraseImmediateBlock()` has been called previously and the job has currently not been finished. The service `Ea_GetStatus()` returns the value `MEMIF_BUSY`.
`MEMIF_BUSY_INTERNAL`	This state is not used by the EA, because no internal operations are performed at all.

Table 2-4   Module states

2.3.2 Job States

Point in Time	Job State	Correlating Module State
After successfully finished job	`MEMIF_JOB_OK`	`MEMIF_IDLE`
After a job has been started	`MEMIF_JOB_PENDING`	`MEMIF_BUSY`
After `Ea_Cancel()`, if previous state was `MEMIF_JOB_PENDING`	`MEMIF_JOB_CANCELLED`	`MEMIF_IDLE`
After a read job has been finished and an invalidated block was found, or no block was found at all (not written yet)	`MEMIF_BLOCK_INVALID`	`MEMIF_IDLE`
After a block is read which has not been written successfully before.	`MEMIF_BLOCK_INCONSISTENT`	`MEMIF_IDLE`
After a job has been finished and retrieving data (independent from management information of payload data) from the EEPROM failed	`MEMIF_JOB_FAILED`	`MEMIF_IDLE`

Table 2-5   Job states

2.4 Main Function

All jobs (Read, Write, InvalidateBlock or EraseImmediateBlock) will be executed asynchronously and are processed by a job state machine. This means that after an asynchronous API call (e.g. `Ea_Write()`) the job and its parameters are internally stored and are being processed by the MainFunction later on.

The function `Ea_MainFunction()` can either be called cyclically with a fixed cycle time by the scheduler or in a background task. The mode of main function triggering is pre-compile configurable via parameter 'EaMainFunctionTriggering'.

In order to check if a job has been finished the upper layer (i.e. NVM) has to call the API service `Ea_GetStatus()` to get the current EA state. After an idle module state the upper layer can request the result of the recently performed job by calling the API service `Ea_GetJobResult()`.

It shall be pointed out that the polling mode described above can also be replaced by callback mechanism if the upper layer module supports this behavior.

> **!** **Caution**
> The EA must be initialized before the services are called.

Only one asynchronous job (Read, Write, InvalidateBlock or EraseImmediateBlock) can be accepted at a time. Therefore, the EA module status must be `MEMIF_IDLE`, before a new job can be requested. The module status can be retrieved by calling the service `Ea_GetStatus()`. Usually, the upper layer module (i.e. NVM) is responsible for synchronizing the pending jobs and assigning it to the EA consecutively, whereas in the Flash Boot Loader mode the Flash Boot Loader application is responsible doing that.

The values of the `BlockNumber` parameter used in most of the API functions are defined by the EA component and no other values should be used. Symbolic names of the blocks are provided to increase the readability.

2.4.1 Processing of a Read Job

The EA provides the service `Ea_Read()` for reading a block. This service reads the data of the block which has been written last.

This asynchronous job is initiated with the API function `Ea_Read()` and is processed by subsequent calls of `Ea_MainFunction()` until the EA returns back to idle state, which means that the job has been finished.

2.4.2 Processing of a Write Job

To write the current block content to the EEPROM, the API function `Ea_Write()` is used. The EA component searches for the next free position in the current block's memory area and requests a job to the underlying EEPROM driver to write the new instance of the block.

This asynchronous job is initiated with the API function `Ea_Write()` and is processed by subsequent calls of `Ea_MainFunction()` until the EA returns back to idle state, which means that the job has been finished.

2.4.3 Processing of an InvalidateBlock Job

The API function `Ea_InvalidateBlock()` is used to invalidate the block content in EEPROM. The EA module marks the block as invalid and reports `MEMIF_BLOCK_INVALID` if such a block is read after the invalidation process.

This asynchronous job is initiated with the API function `Ea_InvalidateBlock()` and is processed by subsequent calls of `Ea_MainFunction()` until the EA returns back to idle state, which means that the job has been finished.

2.4.4 Processing of an EraseImmediateBlock Job

To prepare a block/dataset for an immediate write job, the API service `Ea_EraseImmediateBlock()` shall be used. Hereupon, all bytes of the whole memory area are erased, including management information bytes and data area.

This asynchronous job is initiated with the API function `Ea_EraseImmediateBlock()` and is processed by subsequent calls of `Ea_MainFunction()` until the EA returns back to idle state, which means that the job has been finished.

The intention of this service is to decrease the time for writing a high priority data which usually contains crash data.

2.5 Error Handling

2.5.1 Development Error Reporting

By default, development errors are reported to the DET using the service `Det_ReportError()` as specified in [2], if development error reporting is enabled.

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

The reported service IDs identify the services of the EA which are described in 4.3. The following table presents the service IDs and the related services:

Service ID	Service
0x00	Ea_Init
0x01	Ea_SetMode
0x02	Ea_Read
0x03	Ea_Write
0x04	Ea_Cancel
0x05	Ea_GetStatus
0x06	Ea_GetJobResult
0x07	Ea_InvalidateBlock
0x08	Ea_GetVersionInfo
0x09	Ea_EraseImmediateBlock
0x10	Ea_JobEndNotification
0x11	Ea_JobErrorNotification
0x12	Ea_MainFunction

Table 2-6   Mapping of service IDs to services

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

Error Code		Description
0x01	EA_E_UNINIT	API service (except `Ea_GetStatus()`, `Ea_GetVersionInfo()` and `Ea_Init()`) called although the EA is not initialized yet
0x02	EA_E_INVALID_BLOCK_NO	API service called with invalid block number.
0x03	EA_E_INVALID_BLOCK_OFS	API service `Ea_Read()` called with wrong offset.
0x04	EA_E_INVALID_DATA_POINTER	API service (`Ea_Read()` or `Ea_Write()`) called with null pointer as pointer to data buffer.

Error Code		Description
0x05	EA_E_INVALID_BLOCK_LEN	API service Ea_Read() called with wrong length.
0x06	EA_E_BUSY	API service called when already a job is currently processed.
0x07	EA_E_BUSY_INTERNAL	Not used.
0x08	EA_E_INVALID_CANCEL	API service Ea_Cancel() called when module status is not busy.
0x0A	EA_E_PARAM_MODE	Not used.

Table 2-7   Errors reported to DET

2.5.1.1   Parameter Checking

AUTOSAR requires that API functions check the validity of their parameters. The checks in Table 2-8 are internal parameter checks of the API functions.

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

Service	EA_E_BUSY_INTERNAL	EA_E_UNINIT	EA_E_INVALID_BLOCK_NO	EA_E_INVALID_BLOCK_OFS	EA_E_INVALID_DATA_POINTER	EA_E_PARAM_MODE	EA_E_INVALID_BLOCK_LEN	EA_E_BUSY	EA_E_INVALID_CANCEL
Ea_Init									
Ea_SetMode									
Ea_Read		■	■	■	■		■	■	
Ea_Write		■	■		■			■	
Ea_InvalidateBlock		■	■					■	
Ea_EraseImmediateBlock		■	■					■	
Ea_Cancel		■							■
Ea_GetStatus									
Ea_GetJobResult		■							
Ea_GetVersionInfo					■				
Ea_JobEndNotification									

Check/Error code Service	EA_E_BUSY_INTERNAL	EA_E_UNINIT	EA_E_INVALID_BLOCK_NO	EA_E_INVALID_BLOCK_OFS	EA_E_INVALID_DATA_POINTER	EA_E_PARAM_MODE	EA_E_INVALID_BLOCK_LEN	EA_E_BUSY	EA_E_INVALID_CANCEL
Ea_JobErrorNotification									
Ea_MainFunction		■							

Table 2-8   Development Error Reporting: Assignment of checks to services

Detected development errors are reported to the development error tracer by default. The called service is aborted immediately if an error has been detected. Services with return value of type `Std_ReturnType` return the value `E_NOT_OK`. Services with return type `void` just abort execution of the function and return to the caller.

2.5.2 Production Code Error Reporting

Production code error reporting is not supported in the current version of the EA, because the AUTOSAR specification does not specify any error which shall be reported in production mode.

2.6 Partitions

EA employs a concept of partitions. A partition can be thought of an emulation space that is managed separately from other ones:

- Multiple EEPROM devices / EEPROM drivers are supported by using separate partitions in EA module

- Alignments, such as write and address alignment, can be configured for each partition separately

- Errors in one partition do not affect data in other ones

- Each partition has its own start address and size. But next partition must not start at the end of previous partition. There can be gaps between partitions.

- Block with smallest Block Id has the smallest address in the partition. Each block has to be assigned to a partition.

> **Note**
> EA configurations for FBL and Application do not need to share all partitions. E.g. a partition containing only application data may remain unknown to the FBL. However, shared partitions must refer to identical Eep configurations (EepConfigSet container), and they must match in address and size, as well as in alignment settings.

2.7 Block storage

A configured EA block may consist of several
`/Ea/EaBlockConfiguration/EaNumberOfDatasets` depending on its NvM block configuration as a native, redundant or dataset block.
An EA dataset itself may consist of several
`/Ea/EaBlockConfiguration/EaNumberOfInstances` in order to spread write accesses and to prevent hardware from getting overstressed.
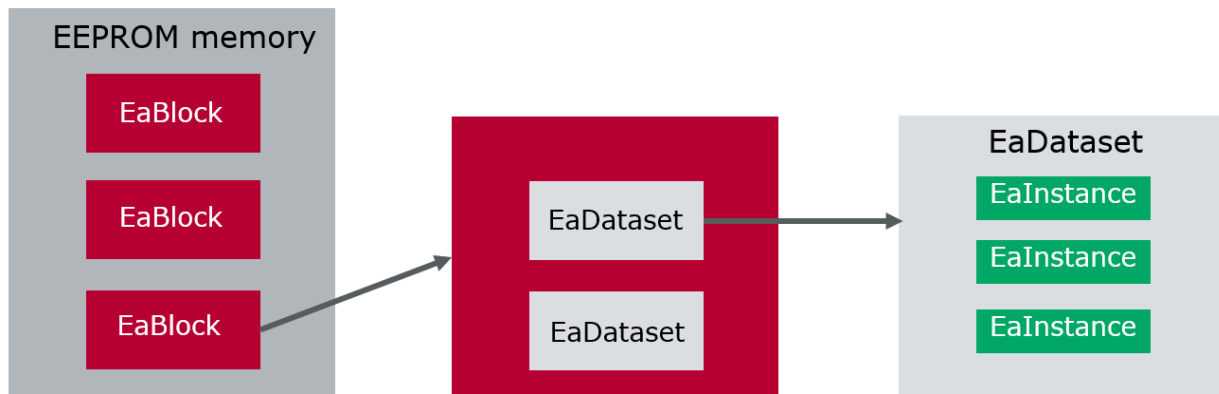
Figure 2-1   Block storage overview

An EA block is always assigned to one partition which determines the location and alignments characteristics of the block (see also 2.6):

All EaInstances will start at addresses being an integral multiple of
`/Ea/EaPartitionConfiguration/EaPartitionAddressAlignment`.
The AddressAlignment itself must be equal or a multiple of
`/Ea/EaPartitionConfiguration/EaPartitionWriteAlignment`.
The WriteAlignment itself must be equal or a multiple of EEPROM driver
`/Eep/EepPublishedInformation/EepWriteUnitSize`.

> **Note**
>
> The `EepWriteUnitSize` does not necessarily reflect the physical EEPROM page size in bytes. EEPROM driver may just simulate to the EA byte wise write access, but EEPROM may be actually written at a higher page size.
>
> Therefore, dependent on the real EEPROM page size and the EA block configuration, EA instances or even EA blocks may not be placed on physical independent EEPROM pages.
>
> If necessary and required, EA blocks can be purposely placed on independent physical EEPROM pages by configuring appropriate EA Partitions.
> For example by setting AddressAlignment of a partition to EEPROM page size or even configure an own Partition for a Block with dedicated StartAddress and Size. Keep in mind when doing this, it can mean that EEPROM is not fully occupied by blocks but there is space left unused between blocks and instances.

3 Integration

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

3.1 Scope of Delivery

The delivery of the EA contains the files which are described in the chapters 3.1.1 and 3.1.2:

3.1.1 Static Files

File Name	Description
Ea.c	Contains the implementation of the interfaces of the EA.
Ea.h	Declares the interface of the EA.
Ea_Cbk.h	Declares the callback functions of the EA.
Ea_PartitionHandler.c	Responsible for partition relevant data.
Ea_PartitionHandler.h	Declares the interface of PartitionHandler.
Ea_BlockHandler.c	Responsible for block relevant data.
Ea_BlockHandler.h	Declares the interface of BlockHandler.
Ea_DatasetHandler.c	Responsible for dataset relevant data.
Ea_DatasetHandler.h	Declares the interface of DatasetHandler.
Ea_InstanceHandler.c	Responsible for instance relevant data.
Ea_InstanceHandler.h	Declares the interface of InstanceHandler.
Ea_TaskManager.c	Responsible for coordinating internal sub-components.
Ea_TaskManager.h	Declares the interface of TaskManager.
Ea_EepCoordinator.c	Provides access to EEPROM driver's services.
Ea_EepCoordinator.h	Declares the interface of EepCoordinator.
Ea_Layer1_Erase.c	Internal layer 1 sub-component for erase jobs
Ea_Layer1_Erase.h	Declares the interface of layer 1 erase sub-component.
Ea_Layer1_Invalidate.c	Internal layer 1 sub-component for invalidation jobs
Ea_Layer1_Invalidate.h	Declares the interface of layer 1 invalidate sub-component.
Ea_Layer1_Read.c	Internal layer 1 sub-component for read jobs.
Ea_Layer1_Read.h	Declares the interface of layer 1 read sub-component.
Ea_Layer1_Write.c	Internal layer 1 sub-component for write jobs.
Ea_Layer1_Write.h	Declares the interface of layer 1 write sub-component.
Ea_Layer2_WriteInstance.c	Internal layer 2 sub-component for writing instances.
Ea_Layer2_WriteInstance.h	Declares the interface of layer 2 write instance sub-component.
Ea_Layer2_InvalidateInstance.c	Internal layer 2 sub-component for invalidating instances.
Ea_Layer2_InvalidateInstan	Declares the interface of layer 2 invalidate instance sub-

File Name	Description
`ce.h`	component.
`Ea_Layer2_InstanceFinder.c`	Internal layer 2 sub-component for finding instances.
`Ea_Layer2_InstanceFinder.h`	Declares the interface of layer 2 instance finder sub-component.
`Ea_Layer3_ReadManagementBytes.c`	Internal layer 3 sub-component for reading management information of instances.
`Ea_Layer3_ReadManagementBytes.h`	Declares the interface of layer 3 read management bytes sub-component.

Table 3-1   Static files

3.1.2   Dynamic Files

The dynamic files are generated by the configuration tool DaVinci Configurator.

File Name	Description
`Ea_Cfg.h`	Contains the static configuration part of this module.
`Ea_Cfg.c`	Contains the link-time configuration of this module.
`Ea_MemMap.h`	Generated header file that contains the memory mapping sections of the Ea module.

Table 3-2   Generated files

3.1.3   Memory Mapping

Delivery with MICROSAR Release < R27:

Ea_MemMap.h is available as a static file.
Memory mapping sections of the Ea module are included within the global MemMap.h file (via Ea_MemMap.inc), which will be included by Ea_MemMap.h.

Delivery with MICROSAR Release >= R27:
Ea_MemMap.h is available as a dynamic file. This header file is generated by an own component "MemMap" (for more information refer to [4]).
Memory mapping sections of the Ea module are directly included within generated Ea_MemMap.h to reduce the build time.

> **Note**
> Own memory mapping sections are used for EA internal write and read buffer. These buffers will be forwarded to the underlying EEPROM driver. In case there are buffer alignment restrictions, the integrator must take care that these MemMap sections are correctly aligned.
> In case of buffer alignment restrictions of the forwarded write buffer, additionally it may be needed that
> `/Ea/EaPartitionConfiguration/EaPartitionWriteAlignment`
> must be set to 1, so that the user buffer is also forwarded aligned.

3.2 Dependencies on SW Modules

3.2.1 AUTOSAR OS

The Main-Function `Ea_MainFunction()` is called by the OS scheduler or BSW Schedule Manager respectively.

3.2.2 DET

The EA depends on the DET (by default) in order to report development errors.

Detection and reporting of development errors can be enabled or disabled by the switch "Development Error Detection" within the EA module.

The DET can be replaced optionally by an equivalent component which is responsible to recognize development errors, if no DET component is available.

3.2.3 EEP

The EEPROM driver provides the access to the underlying hardware. The specific properties of the EEPROM hardware influence the configuration of the EA module.

Its services are called to request a special job to the driver.

3.2.4 Callback Functions

Lower Layer Interaction

The EA offers the usage of callback notification functions for the underlying EEPROM driver to inform the EA that a job has been finished successfully or not. The `Ea_JobEndNotification()` can be called when a job is completed with a positive result and the `Ea_JobErrorNotification()` can be called when a job is cancelled, aborted or has failed.

> **Caution**
> The interaction between EA and the underlying driver does not need to be performed via a notification mechanism. Also polling mode can be chosen if desired.

Upper Layer Interaction

The NVM offers the usage of callback notifications for the EA module to inform the NVM upon finishing an EA job. The `Ea_CbkJobEndNotification()` can be called when a job is completed successfully, otherwise `Ea_CbkJobErrorNotification()` can be called.

Usage of callback notifications can be configured via pre-compile switch. If no callbacks are used, NVM polls current job result from EA.

3.3 Dependencies on HW Modules

The EA is principally hardware independent. Nevertheless, the EEPROM driver has to provide some parameters (defined in the BSWMD file) the EA relies on, e.g. the page size or sector sizes.

The parameter the EA depends on are:

> provided memory size

> the Erase-Unit-Size

> the Write-Unit-Size

> the Read-Unit-Size

> used address type (normally `Eep_AddressType`)

> used length type (normally `Eep_LengthType`)

> supported write cycles

4 API Description

4.1 Interfaces Overview

For an interfaces overview please see Figure 1-2.

4.2 Type Definitions

The EA does not specify any API data types.

4.3 Services provided by EA

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

4.3.1 Ea_Init

Prototype	
void **Ea_Init** (void)	
Parameter	
--	--
Return code	
void	--
Functional Description	
This service initializes the EA module and all needed internal variables.	
The EA module doesn't support any runtime configuration. Hence, a pointer to the configuration structure is not needed by this service.	
The EA does not initialize the underlying EEPROM driver. This shall be done separately, e.g. by the ECUM module.	
Particularities and Limitations	
■ This service is synchronous.	
■ This service is non re-entrant.	
■ This service is always available.	
■ This service shall not be called during a pending job.	
Expected Caller Context	
■ Expected to be called in application context.	

Table 4-1   Ea_Init

4.3.2 Ea_SetMode

Prototype	
void **Ea_SetMode** (MemIf_ModeType Mode)	

Parameter	
Mode	`MEMIF_MODE_SLOW`: Normal read access / normal SPI access (in case of an external EEPROM). See corresponding EEPROM manual for further details.
	`MEMIF_MODE_FAST`: Fast read access / SPI burst access (in case of an external EEPROM). See corresponding EEPROM manual for further details.
Return code	
void	--

Functional Description

This service will not be supported by current implementation of EA module because SetMode handling is not clearly specified and does not have any effect on internal EEPROM drivers.

Particularities and Limitations

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

Expected Caller Context

- Expected to be called in application context.

Table 4-2   Ea_SetMode

4.3.3 Ea_Read

Prototype

`Std_ReturnType` **`Ea_Read`** `(`
`uint16  BlockNumber, uint16 BlockOffset, uint8* DataBufferPtr, uint16`
`  Length)`

Parameter	
BlockNumber	Handle of a logical block (depending on block configuration)
BlockOffset	Read address offset inside the block
DataBufferPtr	Pointer to data buffer
Length	Number of bytes to read
Return code	
E_OK	Read job has been accepted.
E_NOT_OK	Read job has not been accepted.

Functional Description
This service invokes the read processing of the specified block. After the job has been processed by the `Ea_MainFunction()` the requested data has been read and passed to the caller.

> **Info**
> The job processing is asynchronous. Hence, it is necessary to poll the EA about its current status by calling the function `Ea_GetStatus()` as long it is busy after a job has been requested successfully. Finally, the result of the finished job can be retrieved by calling the function `Ea_GetJobResult()`. Accordingly, it is possible to notify the upper layer (usually the NVM) via the callback mechanism, if configured.

Additionally, if development mode is configured, parameter checks are done and in case of failure they are reported to the DET by default with the according service ID and the reason of occurrence (refer to 2.5.1.1).

Particularities and Limitations
■ This service is asynchronous.
■ This service is non re-entrant.
■ This service is always available.

Expected Caller Context
■ Expected to be called in application context.

Table 4-3   Ea_Read

4.3.4   Ea_Write

Prototype
`Std_ReturnType` **`Ea_Write`** `( uint16 BlockNumber, uint8* DataBufferPtr )`

Parameter	
BlockNumber	Handle of a logical block (depending on block configuration)
DataBufferPtr	Pointer to data buffer

Return code	
E_OK	Write job has been accepted.
E_NOT_OK	Write job has not been accepted.

Functional Description
This service invokes the write processing of the specified block. After the job has been processed by the `Ea_MainFunction()` the requested data has been written to the non-volatile memory.

> **Info**
> The job processing is asynchronous. Hence, it is necessary to poll the EA about its current status by calling the function `Ea_GetStatus()` as long it is busy after a job has been requested successfully. Finally, the result of the finished job can be retrieved by calling the function `Ea_GetJobResult()`. Accordingly, it is possible to notify the upper layer (usually the NVM) via the callback mechanism, if configured.

Additionally, if development mode is configured, parameter checks are done and in case of failure they are reported to the DET by default with the according service ID and the reason of occurrence (refer to 2.5.1.1).

Particularities and Limitations
■ This service is asynchronous.
■ This service is non re-entrant.
■ This service is always available.

Expected Caller Context
■ Expected to be called in application context.

Table 4-4   Ea_Write

4.3.5   Ea_Cancel

Prototype	
void **Ea_Cancel** (void)	
Parameter	
--	--
Return code	
void	--
Functional Description	

This service cancels a currently pending job and calls `Eep_Cancel()` to cancel a possibly pending EEPROM driver job.

> **i   Info**
> This service is a synchronous call and does not have to be triggered by the `Ea_MainFunction()`.

The status of the EA will be set to `MEMIF_IDLE` and the job result will be set to `MEMIF_JOB_CANCELLED`, if a job was currently pending.

If the EA is currently `IDLE`, calling this service is without any effect.

Particularities and Limitations
■ This service is synchronous.
■ This service is non re-entrant.
■ This service is always available.

Expected Caller Context
■ Expected to be called in application context.

Table 4-5   Ea_Cancel

4.3.6   Ea_GetStatus

Prototype	
MemIf_StatusType **Ea_GetStatus** (void)	
Parameter	
--	--

Return code	
MEMIF_UNINIT	The EA is currently not initialized -> `Ea_Init()` must be called to use the functionality of the EA.
MEMIF_IDLE	The EA is currently idle -> no asynchronous job pending.
MEMIF_BUSY	The EA is currently busy-> an asynchronous job is currently processed by the EA.
MEMIF_BUSY_INTERNAL	Not used.
Functional Description	
This service returns the current module state of the EA synchronously.	
Particularities and Limitations	
■ This service is synchronous. ■ This service is re-entrant. ■ This service is always available.	
Expected Caller Context	
■ Expected to be called in application context.	

Table 4-6   Ea_GetStatus

4.3.7   Ea_GetJobResult

Prototype	
`MemIf_JobResultType` **`Ea_GetJobResult`** `( void )`	
Parameter	
--	--
Return code	
MEMIF_JOB_OK	The last job has been finished successfully.
MEMIF_JOB_PENDING	The last requested job is waiting for execution or is currently being executed.
MEMIF_JOB_CANCELLED	The last job has been cancelled via the `Ea_Cancel()` service.
MEMIF_JOB_FAILED	The EEPROM driver reported an error or the EA could not achieve the requested job due to hardware errors (e.g. memory cell defects).
MEMIF_BLOCK_INCONSISTENT	The requested block's management information is inconsistent; hence it may contain corrupt data. This result happens if a write- job has not been completed (e.g. due to voltage drop) or if bits flips in the management information bytes occurred and can not be corrected. Furthermore, if write-verify for a specific block is enabled (within the configuration tool) and the comparison of the written data failed the job result will also gain this value.
MEMIF_BLOCK_INVALID	The requested block has been invalidated previously via the service `Ea_InvalidateBlock()` or reading from an erased block is achieved (independent from called `Ea_EraseImmediateBlock()` or a never written block).
Functional Description	
This service returns the result of the last job executed.	

Particularities and Limitations
■ This service is synchronous.
■ This service is re-entrant.
■ This service is always available.
Expected Caller Context
■ Expected to be called in application context.

Table 4-7   Ea_GetJobResult

4.3.8 Ea_InvalidateBlock

Prototype
`Std_ReturnType` **`Ea_InvalidateBlock`** `( uint16 BlockNumber )`

Parameter	
BlockNumber	Number of logical block (depending on block configuration)

Return code	
E_OK	Invalidate job has been accepted.
E_NOT_OK	Invalidate job has not been accepted.

Functional Description
This service invokes the invalidation procedure for the selected block. If the service succeeds the most recent data block is marked as INVALID. **Info** The job processing is asynchronous. Hence, it is necessary to poll the EA about its current status by calling the function `Ea_GetStatus()` as long it is busy after a job has been requested successfully. Finally, the result of the finished job can be retrieved by calling the function `Ea_GetJobResult()`. Accordingly, it is possible to notify the upper layer (usually the NVM) via the callback mechanism, if configured. Additionally, if development mode is configured, parameter checks are done and in case of failure they are reported to the DET by default with the according service ID and the reason of occurrence (refer to 2.5.1.1).

Particularities and Limitations
■ This service is asynchronous.
■ This service is non re-entrant.
■ This service is always available.
Expected Caller Context
■ Expected to be called in application context.

Table 4-8   Ea_InvalidateBlock

4.3.9 Ea_GetVersionInfo

Prototype
`void` **`Ea_GetVersionInfo`** `( Std_VersionInfoType *VersionInfoPtr )`

Parameter	
VersionInfoPtr	Pointer to where to store the version information of this module.
Return code	
void	--
Functional Description	

This service returns the version information of this module. The version information includes:

> Module ID

> Vendor ID

> Vendor specific version numbers.

Additionally, if development mode is configured, parameter checks are done and in case of failure they are reported to the DET with the according service ID and the reason of occurrence (refer to 2.5.1.1).

Particularities and Limitations

- This service is synchronous.
- This service is non re-entrant.
- This service is available, depending on pre-compile configuration checkbox 'Enable Ea_GetVersionInfo API' which is configured within the configuration tool.
- Checking the parameter of the API service for not being a null pointer can be switched off/on at pre-compile time by the configuration tools checkbox 'Check Parameter VersionInfo'.

Expected Caller Context

- Expected to be called in application context.

Table 4-9   Ea_GetVersionInfo

4.3.10 Ea_EraseImmediateBlock

Prototype	
`Std_ReturnType` **`Ea_EraseImmediateBlock`** `( uint16 BlockNumber )`	
Parameter	
BlockNumber	Number of logical block (depending on block configuration)
Return code	
E_OK	Erase job has been accepted.
E_NOT_OK	Erase job has not been accepted.

Functional Description

This service invokes an erase job to provide pre-erased memory areas for blocks with high priority data. The intention of this service is to decrease the time for writing high priority data which usually contains crash data.

The erased block is marked as invalid implicitly. Thus, a subsequent read request on the erased block completes with job result `MEMIF_BLOCK_INVALID`.

> **i** **Info**
> The job processing is asynchronous. Hence, it is necessary to poll the EA about its current status by calling the function `Ea_GetStatus()` as long it is busy after a job has been requested successfully. Finally, the result of the finished job can be retrieved by calling the function `Ea_GetJobResult()`. Accordingly, it is possible to notify the upper layer (usually the NVM) via the callback mechanism, if configured.

Additionally, if development mode is configured, parameter checks are done and in case of failure they are reported to the DET by default with the according service ID and the reason of occurrence (refer to 2.5.1.1).

Particularities and Limitations

- This service is asynchronous.
- This service is non re-entrant.
- This service is always available.

Expected Caller Context

- Expected to be called in application context.

Table 4-10   Ea_EraseImmediateBlock

4.3.11 Ea_MainFunction

Prototype	
void **Ea_MainFunction** (void)	
Parameter	
--	--
Return code	
void	--
Functional Description	

This service triggers the processing of the internal state machine and handles the asynchronous job and management operations.

The complete handling of the job and the detection of invalidated or inconsistent blocks will be done in the internal job state machine.

Additionally, if development mode is configured, the initialized-check is done and in case of failure it is reported to the DET by default with the according service ID (refer to 2.5.1.1).

Particularities and Limitations

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

Expected Caller Context

■ Expected to be called in application context.

Table 4-11   Ea_MainFunction

4.4   Services used by EA

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

Component	API
DET	Det_ReportError
EEP	Eep_Read Eep_Write Eep_Compare Eep_Erase Eep_GetJobResult Eep_GetStatus Eep_SetMode Eep_Cancel
NVM	NvM_JobEndNotification (optionally) NvM_JobErrorNotification (optionally)

Table 4-12   Services used by the EA

4.5   Callback Functions

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

4.5.1   Ea_JobEndNotification

Prototype	
void **Ea_JobEndNotification** (void)	
Parameter	
--	--
Return code	
void	--
Functional Description	
This service can be called by the underlying EEPROM driver to report the successful end of an asynchronous operation.	

Particularities and Limitations

- This callback function is synchronous.
- This callback function is non re-entrant.
- This callback function is only available if the underlying EEPROM driver uses the callback mechanism of the EA.

> **Info**
> If polling mode is configured at pre-compile time this function is not available.

- If the EA is configured (via the configuration tool) to poll the underlying EEPROM driver for its current status (via `Eep_GetStatus()`) and the corresponding job result (via `Eep_GetJobResult()`) this callback function is not available. The configuration for the availability of this function is done at pre-compile time. Accordingly, if the notification mechanism shall be used for the interaction between EA and EEPORM driver, this callback function is provided by the EA.

Expected Caller Context

- This function might be called on interrupt level, depending on the calling function.

Table 4-13   Ea_JobEndNotification

4.5.2   Ea_JobErrorNotification

Prototype
void **Ea_JobErrorNotification** (void)

Parameter	
--	--

Return code	
void	--

Functional Description
This routine can be called by the underlying EEPROM driver to report the failure of an asynchronous operation.

Particularities and Limitations

- This callback function is synchronous.

- This callback function is non re-entrant.

- This callback function is only available if the underlying EEPROM driver uses the callback mechanism of the EA.

> **Info**
> If polling mode is configured at pre-compile time this function is not available.

- If the EA is configured (via the configuration tool) to poll the underlying EEPROM driver for its current status (via `Eep_GetStatus()`) and the corresponding job result (via `Eep_GetJobResult()`) this callback function is not available. The configuration for the availability of this function is done at pre-compile time. Accordingly, if the notification mechanism shall be used for the interaction between EA and EEPORM driver, this callback function is provided by the EA.

Expected Caller Context

- This function might be called on interrupt level, depending on the calling function.

Table 4-14   Ea_JobErrorNotification

4.6   Configurable Interfaces

API	Description
Function `Ea_GetVersionInfo()`	This function can be enabled/disabled by the configuration switch 'Version Info API'.

Functions
`Ea_JobEndNotification()`
`Ea_JobErrorNotification()`
	The functions can be enabled/disabled by the configuration switch 'Polling Mode'.

5 Configuration

5.1 Configuration Variants

- VARIANT-PRE-COMPILE

The configuration classes of the EA parameters depend on the supported configuration variants. For their definitions please see the Ea_bswmd.arxml file.

EA can be configured using the following tool:

- DaVinci Configurator 5 (AUTOSAR 4 packages only). Parameters are explained within the tool.

The outputs of the configuration and generation process are the configuration source files.

6 Glossary and Abbreviations

6.1 Glossary

Term	Description
DaVinci Configurator 5	Generation tool for MICROSAR components
Block	Logical entity with certain data length provided by NVM or FBL.
Dataset	The Dataset NVRAM block is an array of equally sized data blocks (NV/ROM). **Example** Datasets can be used to provide different configuration settings for the CAN driver (CAN IDs, filter settings, …) to an ECU which has otherwise identical application software (e.g. door module). As a further example, it is possible to provide multiple driver seat configuration profiles via datasets.
Instance	Datasets may consist of several instances in order to spread erase/write accesses and to prevent hardware from getting overstressed.

Table 6-1   Glossary

6.2 Abbreviations

Abbreviation	Description
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BSW	Basis Software
DEM	Diagnostic Event Manager
DET	Development Error Tracer
EA	EEPROM Abstraction
ECU	Electronic Control Unit
ECUM	ECU Manager
EEPROM	Electrically Erasable Programmable Read Only Memory
FBL	Flash Bootloader
HIS	Hersteller Initiative Software
MEMIF	Memory Abstraction Interface Module
MICROSAR	Microcontroller Open System Architecture (the Vector AUTOSAR solution)
NVM	NVRAM Manager

NVRAM	Non Volatile Random Access Memory
OS	Operating System
SRS	Software Requirement Specification
SWS	Software Specification

Table 6-2   Abbreviations

7   Contact

Visit our website for more information on

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

www.vector-informatik.com