VECTOR

MICROSAR vMem

Technical Reference

Version 1.03.00

Authors	Bernhard Karl, Tomas Ondrovic, Charlotte Ricke
Status	Released

Document Information

History

Author	Date	Version	Remarks
Bernhard Karl, Tomas Ondrovic	2018-04-23	1.00.00	Initial creation
Charlotte Ricke	2018-06-21	1.01.00	Core and LowLevel part implemented. Updated Chapter 4.1 and 3.5. Also, minor changes in Chapter 3.3.
Charlotte Ricke	2018-08-06	1.02.00	Updated Chapter 3.1, 3.2 and 3.5.
Bernhard Karl	2018-09-25	1.03.00	Support of burst functionality added inside Chapter 3.1. Wording: ”FOTA” was replaced by “OTA” and “Software Download” in Chapter 2.

Reference Documents

No.	Source	Title	Version
[1]	AUTOSAR	AUTOSAR_SWS_DevelopmentErrorTracer.pdf	3.2.0
[2]	Vector	TechnicalReference_vMemAccM.pdf	2.x.x
[3]	AUTOSAR	AUTOSAR_ SWS_FlashDriver.pdf	3.2.0

Scope of the Document

This technical reference describes the general use of the vMem core functionality.
All aspects which are hardware and flash controller specific (Low Level part of vMem) are described in the additional document of the vMem instance, which is also part of the 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 Component History

The component history gives an overview over the important milestones that are supported in the different versions of the component.

Component Version	New Features
1.00.00	Creation of component
1.01.00	Core and LowLevel concept is now implemented
1.02.00	Improvements in Core generator
1.03.00	Improvements to obtain quality status QM
1.04.00	Write and erase burst functionality is now supported by Core

Table 1-1 Component history

2 Introduction

This document describes the core functionality, API and configuration of the MICROSAR BSW module vMem.

Supported AUTOSAR Release:	4	
Supported Configuration Variants:	pre-compile	
Vendor ID:	VMEM_VENDOR_ID	30 decimal (= Vector-Informatik, according to HIS)
Module ID:	VMEM_MODULE_ID	255 decimal

The vMem (Vector Memory Driver) provides services for accessing the NV (non volatile) Memory. The following document describes the core functionality, which every instance of the vMem has realized.

The vMem offers basic functionalities like read, write, erase and user requested status reporting comparable to a classical FLS driver [3]. The vMem was originally created within the Vector solution for Software Download (also known as OTA), which is also its most likely use case. This means that not only the classical AUTOSAR memory stack is a user of the vMem, but also the OTA stack. The two stacks are coordinated in the layer above by the module vMemAccM (Memory Access Manager, for documentation see [2]).

Every vMem driver instance may offer additional hardware specific functionality, supplementing the features specified within this document. These additional services are described in the technical reference of the individual vMem driver instance.

In a Software Download solution more than one vMem instance could be used by the vMemAccM. This depends on the hardware and the use case. For example, there could be separated vMem instances needed for the PFLASH and DFLASH of the device and an additional vMem for an external device.

There are two variants how the vMem can be realized:

- vMem acts as an own Memory Driver.
- vMem acts as a wrapper to a classical Fls Driver. Hence in a OTA environment a Fls driver could be provided without offering special services. In this case OTA functionalities could be executed by a Flash Bootloader.

2.1 Architecture Overview

Note that vMem is not an AUTOSAR module. Therefore, it would be considered as part of the complex driver.

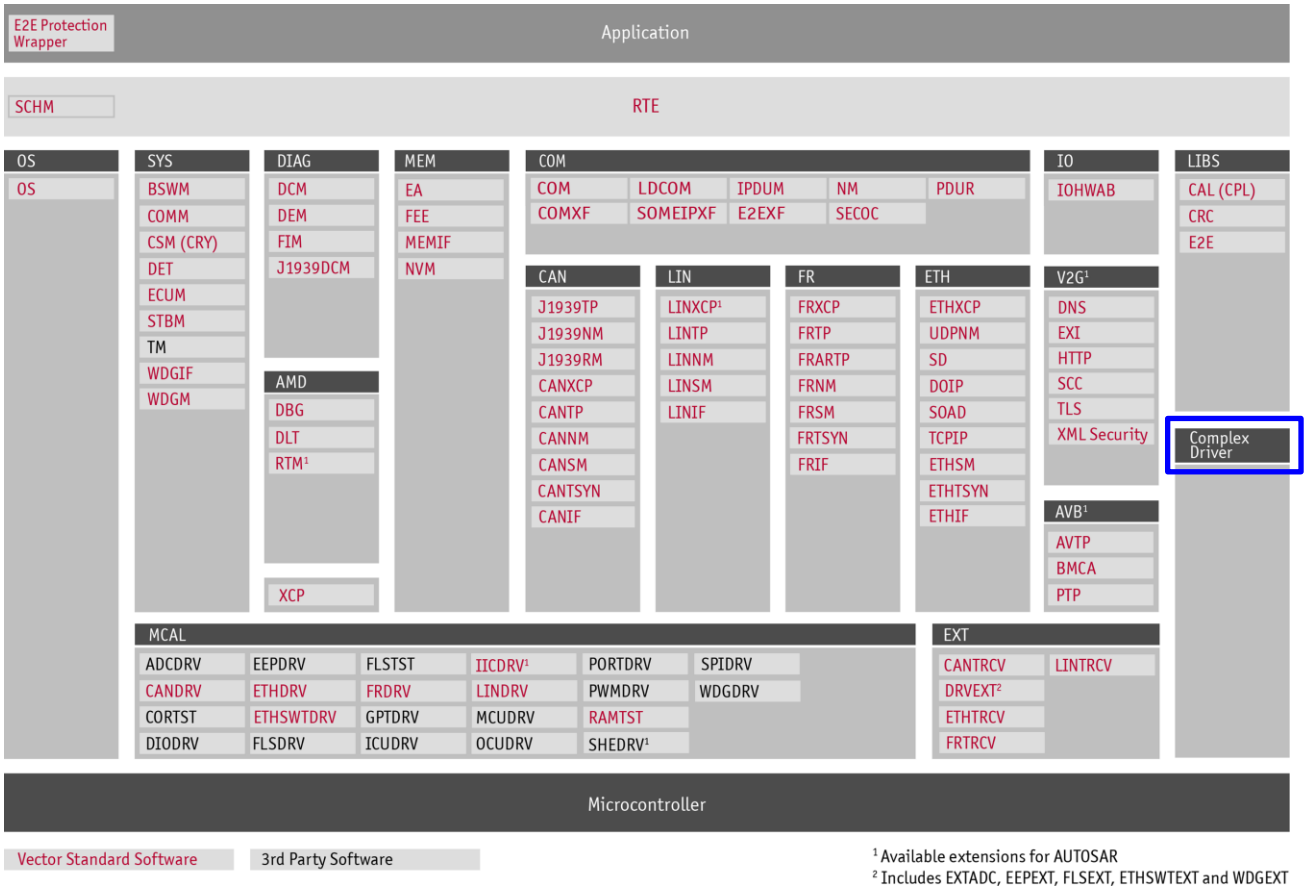

Figure 2-1 AUTOSAR 4.x Architecture Overview

The following figure shows a possible location of vMem in relation to the AUTOSAR memory stack and the OTA memory stack.

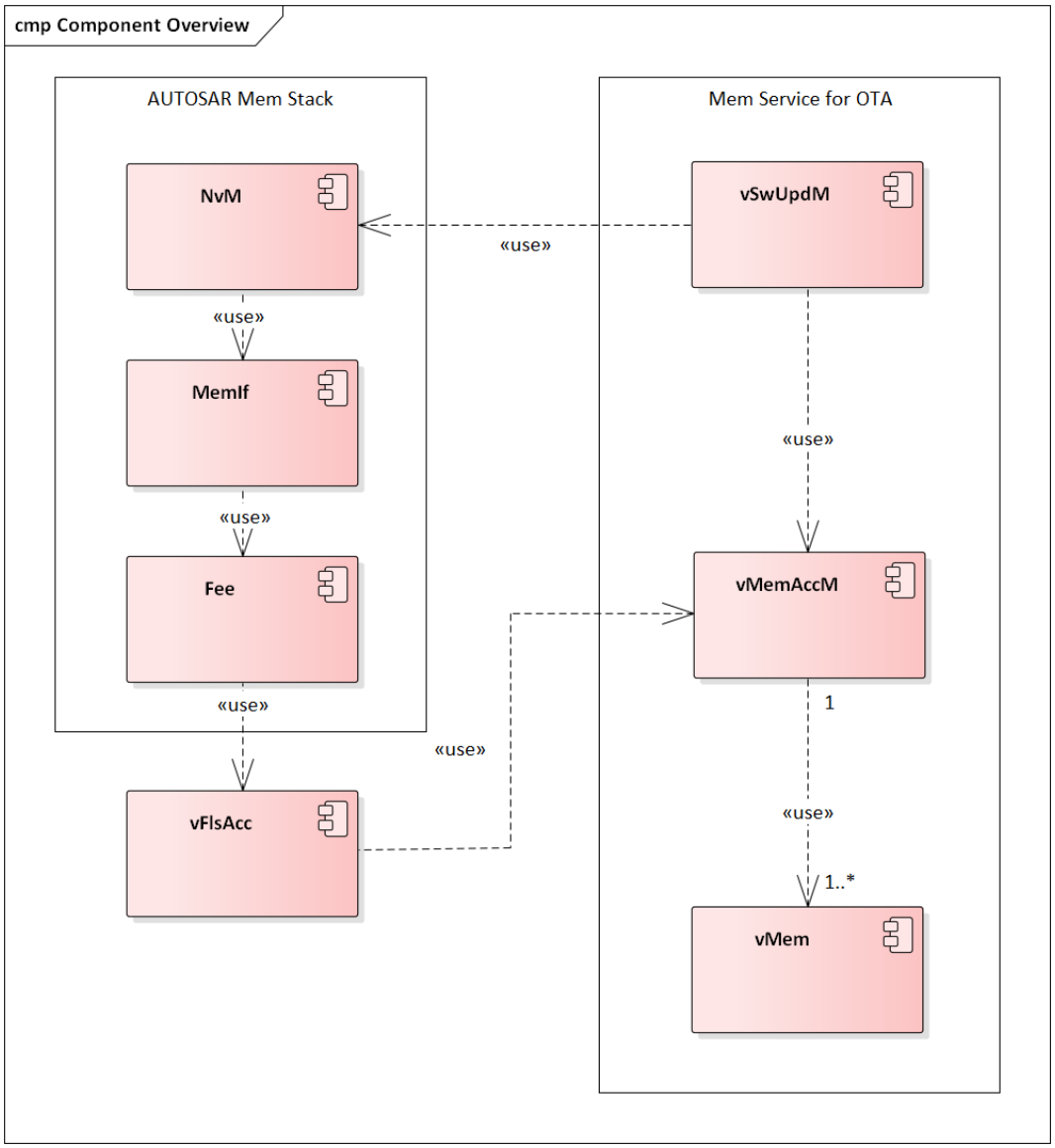

Figure 2-2 Component Overview of one most likely use case

2.2 Component Interfaces

The next figure shows the interfaces to adjacent modules of the vMem. These interfaces are described in chapter 5.

As described in the introduction two types of vMem implementation could be distinguished. In case vMem is a full implemented module:

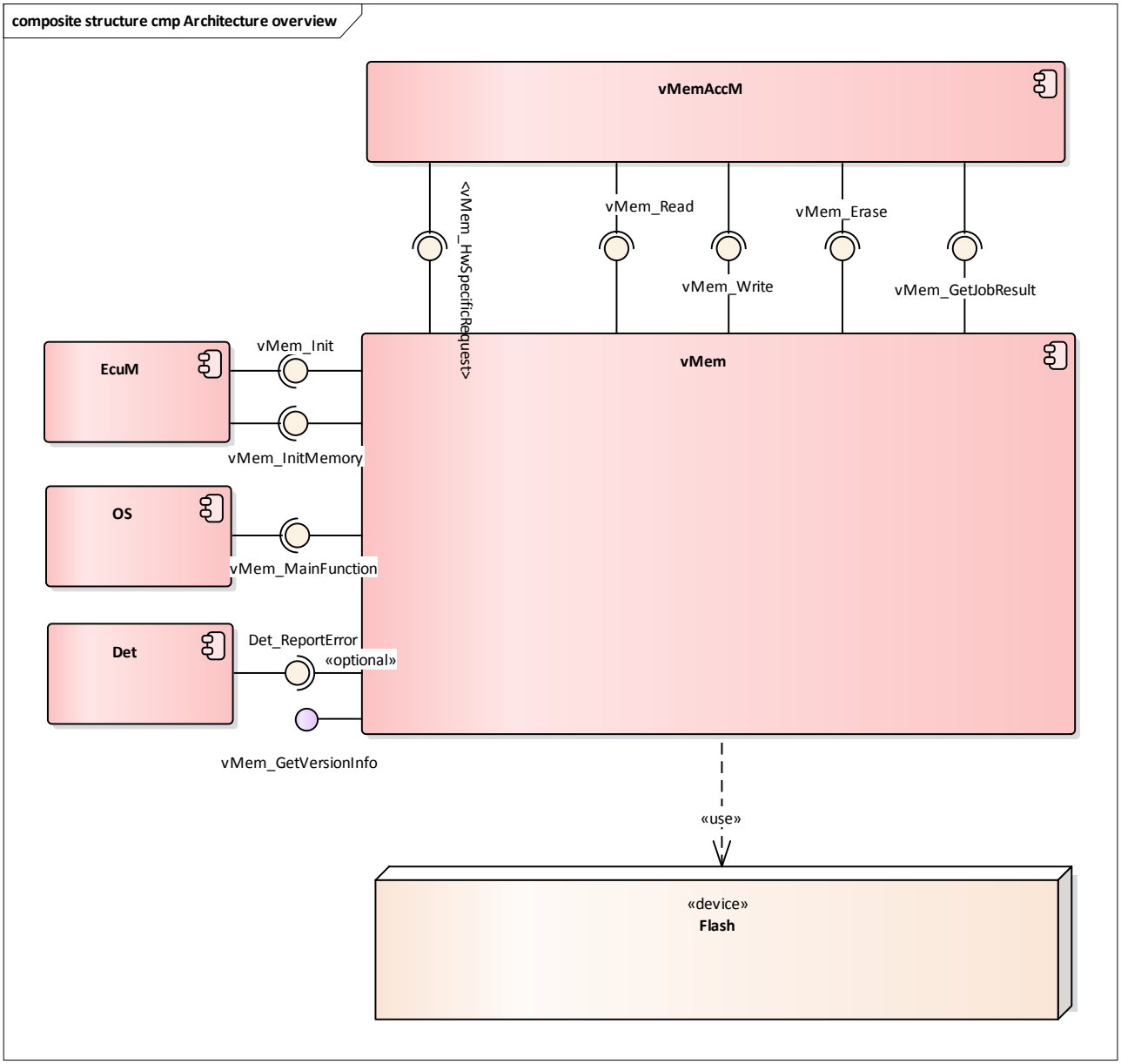

Figure 2-3 Interfaces to adjacent modules of the vMem

In case the vMem acts as an adapter to a Fls Driver:

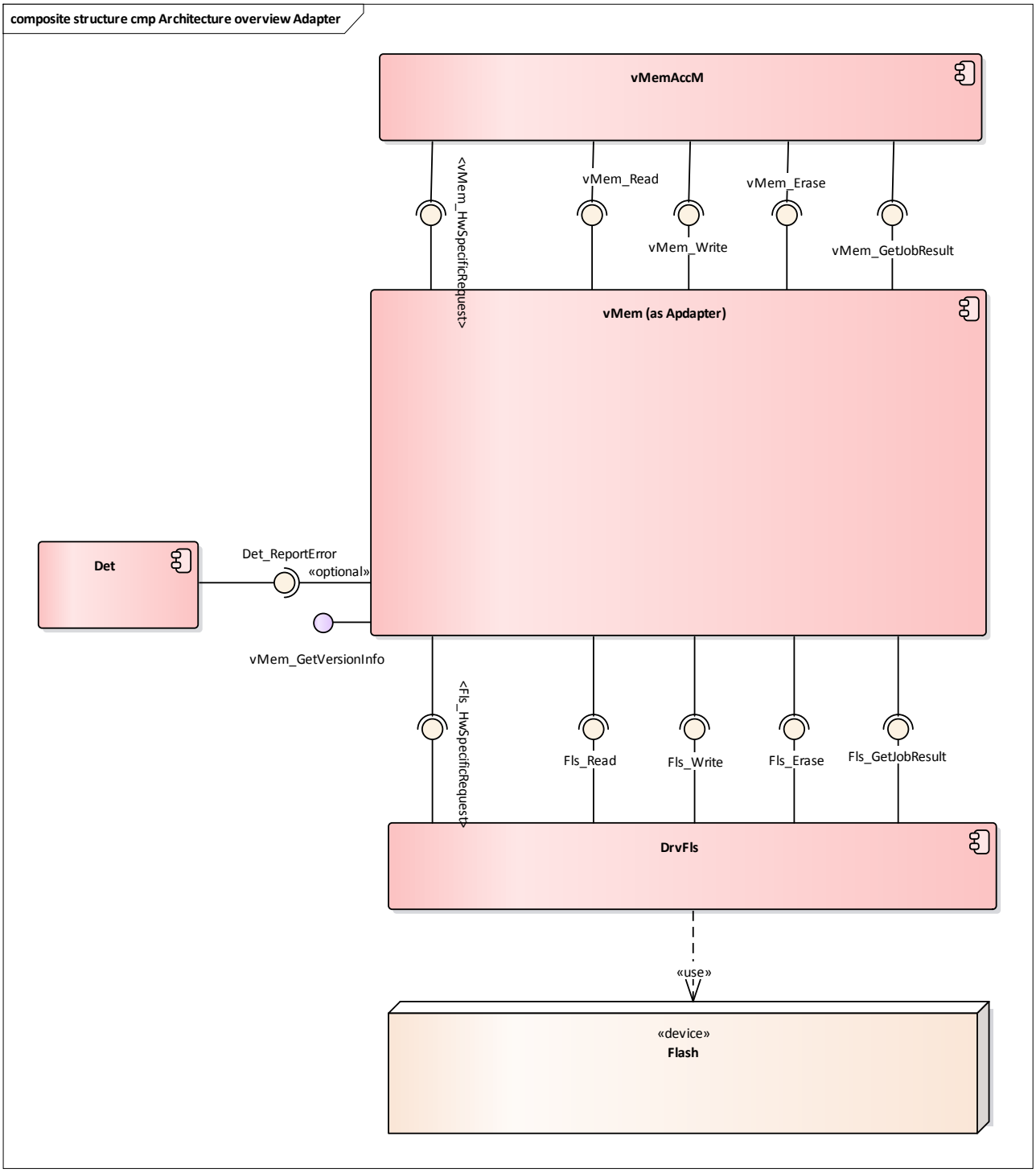

Figure 2-4 Interfaces to adjacent modules of the vMem as an Adapter

3 Functional Description

3.1 Features

The features listed in the following table covers the complete functionality specified for the vMem core. Since the vMem is not an AUTOSAR component all features provided are beyond AUTOSAR standard:

Provided Features
Read data from flash. (optional)
Write data with length *PageSize* or *WriteBurstSize* to flash. (optional)
Erase data with length *SectorSize* or *EraseBurstSize* in flash. (optional)
Provide the current job result.
Performing hardware specific functionality. (optional)
Provide module version information.
Driver initialization. (optional)

Table 3-1 Provided Features

In most cases, the services to read, write or erase data from/to flash will be provided. However, it is possible that a concrete vMem is only used to read data from flash. In that special case, only the read service will be provided. However, the corresponding functions will always be implemented for compatibility reasons. If the service is not supported, the function will return E_NOT_OK when called. It will be documented in the technical reference of the concrete vMem, if one of the memory access services is not provided. There it is also described, if the concrete vMem can write and erase data not only with standard length, but also with configurable burst length.

3.2 Initialization

A vMem is initialized via the services `vMem_InitMemory()` and `vMem_Init()`. The internal variable structures only need to be initialized via the service `vMem_InitMemory()`, in case they are not initialized by the startup code.

The EcuM usually takes care of initialization. If no EcuM is used these functions have to be called by application in correct order. The vMem can be re-initialized by calling the service `vMem_Init()` again, if no job is currently processing.

Caution

If the vMem acts as an adapter there may be no initialization services. In particular, the vMem does not initialize the underlying Fls driver. In that case the underlying Fls driver needs to be initialized before using the vMem

3.3 Main Functions

The vMem has one central processing function `vMem_MainFunction()`. The triggering of this function is described in the technical reference of the vMem instance.

When an asynchronous request is accepted, the job processing will be handled within the `vMem_MainFunction()`.

With the module vMemAccM on top it is secured that the vMem will get at most one asynchronous job request at a time. The services `vMem_Read()`, `vMem_Write()` and `vMem_Erase()` are asynchronous jobs and their result can be polled via the synchronous service `vMem_GetJobResult()`. However, it is possible, depending on the hardware specific implementation of these services, that e.g. the read job is completed, directly when the `vMem_Read()` service returns.

Caution

If vMem acts as an adapter there is no need to use the main function. All necessary jobs are performed by the underlying Fls driver.

Note

The `vMem_MainFunction()` should not be triggered slower than `vMemAccM_MainFunction()`.

3.4 Accessing Functionality

vMem provides the APIs `vMem_Read()`, `vMem_Write()`, `vMem_Erase()` and `vMem_GetJobResult()` via a function pointer table. The vMem instances can be accessed via their instance identifier, passed to the API as parameter.

The user (vMemAccM) retrieves the name and entries of this table from the configuration and can call the respective service through the known entries and instance identifier.

3.5 Error Handling

3.5.1 Development Error Reporting

By default, development errors are reported to the DET using the service `Det_ReportError()` as specified in [1], if development error reporting is enabled (i.e. pre-compile parameter `VMEM_DEV_ERROR_DETECT==STD_ON`).

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

The reported vMem ID is 255.

The reported service IDs identify the services which are described in 5.2. The following table presents the service IDs and the related services:

Service ID	Service
0x00	vMem_Init ()
0x01	vMem_InitMemory ()
0x02	vMem_Read ()
0x03	vMem_Write ()
0x04	vMem_Erase ()
0x05	vMem_MainFunction ()
0x0A	vMem_Get_Job_Result ()
0x10	vMem_GetVersionInfo ()

Table 3-2 Service IDs

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

Error Code		Description
0x0A	VMEM_E_PARAM_CONFIG	API service called with wrong config parameter
0x0B	VMEM_E_PARAM_POINTER	API service used with invalid pointer parameter (NULL)
0x0C	VMEM_E_PARAM_ADDRESS	API service used with invalid address parameter (optional)
0x0D	VMEM_E_PARAM_LENGTH	API service used with invalid length parameter (optional)
0x0E	VMEM_E_PARAM_BUFFER_ALIGNMENT	API service used with invalid buffer parameter (optional)
0x10	VMEM_E_UNINIT	API service used without module initialization
0x11	VMEM_E_INITIALIZATION_FAILED	No or invalid communication towards the hardware during (re-)initialization.
0x12	VMEM_E_PARAM_INSTANCE_ID	API service used with invalid instance identifier parameter.
0x13	VMEM_E_PENDING	The requested instance is already pending.

Table 3-3 Errors reported to DET

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

Service \ Check/Error code	VMEM_E_PARAM_CONFIG	VMEM_E_PARAM_POINTER	VMEM_E_PARAM_ADDRESS	VMEM_E_PARAM_LENGTH	VMEM_E_PARAM_BUFFER_ALIGNMENT	VMEM_E_UNINIT	VMEM_E_INITIALIZATION_FAILED	VMEM_E_PARAM_INSTANCE_ID	VMEM_E_PENDING
vMem_InitMemory()									
vMem_Init()							■		■
vMem_Read()		■	■	■		■		■	■
vMem_Write()		■	■	■		■		■	■
vMem_Erase()			■	■		■		■	■
vMem_MainFunction()						■			
vMem_GetJobResult()						■		■	
vMem_GetVersionInfo()		■							

Table 3-4 Development Error Reporting: Assignment of checks to services

4 Integration

This chapter gives necessary information for the integration of the MICROSAR vMem into an application environment of an ECU. The scope of delivery is described also in the related documentation of the vMem instance.

4.1 Scope of Delivery

The delivery of the vMem contains at least the files which are described in the chapters 4.1.1 and 4.1.2. Additional files that may be needed for the hardware specific implementation are described in the technical reference of the respective vMem. The placeholder <Hw> will be replaced in a concrete instance of the vMem:

4.1.1 Static Files

File Name	Description
vMem_30_<Hw>.c	This is the source file of the vMem. It contains the implementation of the vMem core, that is the part of the implementation, which is the same for all vMem instances.
vMem_30_<Hw>.h	This is the header file of the vMem. It declares the generic interfaces of the vMem.
vMem_30_<Hw>_Types.h	This file defines general types used by vMem and the defines for development error reporting.
vMem_30_<Hw>_LL.c	This is the source file of the vMem LowLevel part. It contains the hardware specific implementation parts of the vMem.
vMem_30_<Hw>_LL.h	This is the header file of the vMem LowLevel part. It declares the interfaces of the vMem LowLevel part, which are used by the vMem core.
vMem_30_<Hw>_Int.h	This internal header file of the vMem declares the common vMem API Read, Write, Erase and GetJobResult. These API can be called by other modules via the FunctionPointerTable, which is maintained by the vMem LowLevel.

Table 4-1 Static files

4.1.2 Dynamic Files

The dynamic files are generated by the configuration tool DaVinci Configurator 5.

File Name	Description
vMem_30_<Hw>_Lcfg.c	This file contains configuration parameters.
vMem_30_<Hw>_Cfg.h	This file contains the public configuration parameters of the vMem.

Table 4-2 Generated files

5 API Tables

For an interfaces overview please see Figure 2-3.

5.1 Type Definitions

The types defined by the vMem are described in this chapter. The initial type definitions are implemented in the module vMemAccM [2].

Type Name	C-Type	Description	Value Range
vMem_ConstDataPtr Type	void*	Type for pointer to buffer passed by user in write request.	Must not be NULL pointer
vMem_DataPtrType	void*	Type for pointer to buffer passed by user in read request.	Must not be NULL pointer.
vMem_JobResultType	enum	Type for the asynchronous job results.	`VMEM_JOB_OK` The last asynchronous job has been finished successfully.
			`VMEM_JOB_PENDING` An asynchronous job is currently being processed.
			`VMEM_READ_CORRECTED_ERRORS` Hardware reported that ECC errors were corrected.
			`VMEM_READ_UNCORRECTABLE_ERRORS` Uncorrectabke ECC errors occurred during read.
			`VMEM_JOB_FAILED` Job failed for some unspecific reason.
vMem_AddressType	uint32	Type for the address of a physical Flash device.	0 – 4294967295
vMem_LengthType	uint32	Type for the length of an asynchronous job.	0 – 4294967295
vMem_InstanceIdType	uint32	Type of the instance identifier.	0 – 4294967295

Table 5-1 Type definitions

5.2 Services provided by vMem

The AUTOSAR related read, write, erase and getJobResult services of the vMem are accessible by a FunctionPointerTable (see also Chapter 3.4). If the vMem acts as an adapter the services inside chapter 5.2.1, 5.2.2 and 5.2.3 are not realized.

5.2.1 vMem_InitMemory

Prototype	
`void vMem_InitMemory (void)`	
Parameter	
void	none
Return code	
void	none
Functional Description	
Function for *_INIT_*-variable initialization. Service to initialize module global variables at power up. This function initializes the variables in *_INIT_* sections. Used in case they are not initialized by the startup code.	
Particularities and Limitations	
-	
Call context	
> TASK (usually startup sequence) > This function is Synchronous > This function is Non-Reentrant	

Table 5-2 vMem_InitMemory

5.2.2 vMem_Init

Prototype	
`void vMem_Init (const vMem_ConfigType *ConfigPtr)`	
Parameter	
ConfigPtr [in]	Pointer to the configuration data (ignored).
Return code	
void	none
Functional Description	
Initialization function. This function initializes the module vMem. It initializes all variables and sets the module state to initialized.	

Particularities and Limitations
> Interrupts are disabled. Module is uninitialized. `vMem_InitMemory` has been called unless `vMem_ModuleInitialized` is initialized by startup code.
Call context
> TASK (usually ECU initialization sequence e.g. EcuM driver Init-phases) > This function is Synchronous > This function is Non-Reentrant

Table 5-3 vMem_Init

5.2.3 vMem_MainFunction

Prototype	
`void` **`vMem_MainFunction`** `(void)`	
Parameter	
void	none
Return code	
void	none
Functional Description	
Monitors and controls the continuous execution of the vMem job processing.	
Particularities and Limitations	
-	
Call context	
> TASK > This function is Synchronous > This function is Non-Reentrant	

Table 5-4 vMem_MainFunction

5.2.4 vMem_GetVersionInfo

Prototype	
`void` **`vMem_GetVersionInfo`** `(Std_VersionInfoType *VersionInfo)`	
Parameter	
VersionInfo [out]	Pointer to where to store the version information. Parameter must not be NULL.
Return code	
void	None

Functional Description
Returns the version information. `vMem_GetVersionInfo()` returns version information, vendor ID and AUTOSAR module ID of the component.
Particularities and Limitations
VMEM_VERSION_INFO_API == STD_ON
Call context
> TASK > This function is Synchronous > This function is Reentrant

Table 5-5 vMem_GetVersionInfo

5.2.5 vMem_Read

Prototype	
`Std_ReturnType` **`vMem_Read`** `(vMem_InstanceIdType InstanceId, vMem_AddressType SourceAddress, vMem_DataPtrType TargetAddressPtr, vMem_LengthType Length)`	
Parameter	
InstanceId [in]	ID of the related vMem instance.
SourceAddress [in]	Address from which the read job shall start.
TargetAddressPtr [out]	Pointer to where the read data shall be stored.
Length [in]	Number of bytes that shall be read.
Return code	
`Std_ReturnType`	E_OK job accepted, E_NOT_OK otherwise
Functional Description	
Reads NV memory content of the related address to passed buffer – from address with length number of bytes. The job status can be polled via `vMem_GetJobResult`.	
Particularities and Limitations	
Expected Caller Context	
> TASK > This function is asynchronous.	

Table 5-6 vMem_Read

5.2.6 vMem_Write

Prototype
`Std_ReturnType` **`vMem_Write`** `(vMem_InstanceIdType InstanceId, vMem_AddressType TargetAddress, vMem_ConstDataPtrType SourceAddressPtr, vMem_LengthType Length)`

Parameter	
InstanceId [in]	ID of the related vMem instance.
TargetAddress [in]	Address from which write job shall start.
SourceAddressPtr [in]	Pointer to the data which shall be written to memory.
Length [in]	Number of bytes that shall be written.
Return code	
Std_ReturnType	E_OK job accepted, E_NOT_OK otherwise
Functional Description	
Writes the passed data to NV memory of the related address – from address with length number of bytes. The length could be either *PageSize* or if burst is supported also *WriteBurstSize.* The job status can be polled via `vMem_GetJobResult`.	
Particularities and Limitations	
Expected Caller Context	
> TASK > This function is asynchronous.	

Table 5-7 vMem_Write

5.2.7 vMem_Erase

Prototype	
`Std_ReturnType` **`vMem_Erase`** `(vMem_InstanceIdType InstanceId, vMem_AddressType TargetAddress, vMem_LengthType Length)`	
Parameter	
InstanceId [in]	ID of the related vMem instance.
TargetAddress [in]	Address from which erase job shall start.
Length [in]	Number of bytes that shall be erased.
Return code	
Std_ReturnType	E_OK job accepted, E_NOT_OK otherwise
Functional Description	
Erases NV memory content of the related address with length number of bytes. The length could be either *SectorSize* or if burst is supported also *EraseBurstSize*. The job status can be polled via `vMem_GetJobResult`.	
Particularities and Limitations	
Expected Caller Context	
> TASK > This function is asynchronous.	

Table 5-8 vMem_Erase

5.2.8 vMem_GetJobResult

Prototype	
`vMem_JobResultType` **`vMem_GetJobResult`** `(vMem_InstanceIdType InstanceId)`	
Parameter	
InstanceId [in]	ID of the related vMem instance.
Return code	
`vMem_JobResultType`	Most recent job result.
Functional Description	
Returns the job result.	
Particularities and Limitations	
Expected Caller Context	
> TASK > This function is synchronous.	

Table 5-9 vMem_GetJobResult

5.3 Services used by vMem

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

If the vMem is used as an adapter to a Fls Driver the services of the corresponding Driver are used, otherwise these are implemented in the vMem itself.

Component	API
DET	Det_ReportError

Table 5-10 Services used by the vMem

6 Configuration

In the vMem the attributes can be configured according to/ with the following methods/ tools:

- DaVinci Configurator 5 (recommended)
- AUTOSAR Generic Configuration Editor (GCE)

6.1 Configuration Variants

The vMem supports the configuration variants

- `VARIANT-PRE-COMPILE`

6.2 Configuration with DaVinci Configurator

The configuration parameters are generally well described within the tool and the vMem_base_bswmd.arxml file.

7 Glossary and Abbreviations

7.1 Glossary

Term	Description
Flash Bootloader	Flash Bootloader gives you a universal and compact solution for reprogramming ECUs quickly, efficiently and securely.

Table 7-1 Glossary

7.2 Abbreviations

Abbreviation	Description
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BSW	Basis Software
DET	Development Error Tracer
DFLASH	Data Flash
ECU	Electronic Control Unit
EcuM	ECU Manager
Fee	Flash EEPROM Emulation
HSM	Hardware Security Module
ISR	Interrupt Service Routine
MemIf	Memory Abstraction Interface
MICROSAR	Microcontroller Open System Architecture (the Vector AUTOSAR solution)
NV, NVRAM	Non Volatile Random Access Memory
NvM	NVRAM Manager
OTA	Over The Air
OTP	One Time Programmable
PFlash	Programm Flash
vFlsAcc	Vector Flash Access (Adapter between AUTOSAR memory stack and other components, e.g. the OTA memory stack)
vMem	Vector Memory (Driver)
vMemAccM	Vector Memory Access Manager
vSwUpdM	Vector Software Update Manager

Table 7-2 Abbreviations

8 Contact

Visit our website for more information on

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

www.vector.com