Document Title	Guide to BSW Distribution
Document Category	Explanation (EXP)
Document UID	631
Document Status	published
Part of AUTOSAR Standard	Classic Platform
Part of Standard Release	R26-11

Document Change History

Date	Release	Changed by	Description
2026-11-26	R26-11	AUTOSAR Release Management	• Updated description of large data path from LdCom to Com • Added constraint for asynchronous DMA copy operations • Added hints for configuring the BswM per partition and multi-core use of BSW modules
2025-11-27	R25-11	AUTOSAR Release Management	• Added multi-partition service component architecture pattern • Added explanation of EcucBusToPartitionMapping
2024-11-27	R24-11	AUTOSAR Release Management	• Removed partition restart
2023-11-23	R23-11	AUTOSAR Release Management	• Cleanup of outdated parameter references
2022-11-24	R22-11	AUTOSAR Release Management	• Updated multicore type for CanXL and Wdg
2021-11-25	R21-11	AUTOSAR Release Management	• Clarified partition scope of MCAL • Removed restriction for BSW partitions per core
2020-11-30	R20-11	AUTOSAR Release Management	• Added chapter on crypto-stack distribution
2019-11-28	R19-11	AUTOSAR Release Management	• Incorporation of concept "BSW Multicore Distribution" • Changed Document Status from Final to published

△

2018-10-31	4.4.0	AUTOSAR Release Management	• Incorporation of concept "MCAL Multicore Distribution"
2017-12-08	4.3.1	AUTOSAR Release Management	• Editorial changes
2016-11-30	4.3.0	AUTOSAR Release Management	• Editorial changes
2014-10-31	4.2.1	AUTOSAR Release Management	• Incorporation of concept "Mechanisms and constraints to protect ASIL BSW against QM BSW" • Minor clarifications
2014-03-31	4.1.3	AUTOSAR Release Management	• Clarified terms
2013-03-15	4.1.1	AUTOSAR Administration	• Initial release

Disclaimer

This work (specification and/or software implementation) and the material contained in it, as released by AUTOSAR, is for the purpose of information only. AUTOSAR and the companies that have contributed to it shall not be liable for any use of the work.

The material contained in this work is protected by copyright and other types of intellectual property rights. The commercial exploitation of the material contained in this work requires a license to such intellectual property rights.

This work may be utilized or reproduced without any modification, in any form or by any means, for informational purposes only. For any other purpose, no part of the work may be utilized or reproduced, in any form or by any means, without permission in writing from the publisher.

The work has been developed for automotive applications only. It has neither been developed, nor tested for non-automotive applications.

The word AUTOSAR and the AUTOSAR logo are registered trademarks.

Table of Contents

1 Introduction 6

2 BSW Distribution in Multi-Core Systems 7
2.1 Overview . . . 7
2.1.1 Supported Scenarios . . . 7
2.1.2 Performance Use Cases and Hardware Assigned to Different Cores 8
2.1.3 Technical Overview . . . 8
2.2 Parallel Execution of BSW modules . . . 11
2.2.1 Core-Dependent Branching . . . 11
2.2.2 Master/Satellite-approach . . . 11
2.2.3 Using the BSW Scheduler for Inter-Partition-Communication . . . 13
2.2.4 Using Shared Buffers (in systems without memory protection) . . . 14
2.2.5 Accessing Hardware/Drivers . . . 16
2.2.6 Concurrency safe implementation of modules . . . 16
2.2.7 Kernel based Master-Satellite Realization . . . 17
2.2.8 Atomic Operations Library . . . 20
2.3 SchM Interfaces for Parallel BSW execution . . . 20
2.4 Configuration of Basic Software in Partitioned Systems . . . 21
2.4.1 Task Mapping . . . 21
2.4.2 General Configuration of Master and Satellites . . . 25
2.4.3 Configuring the BswM (per Partition) . . . 25
2.4.4 Configuring the EcuM (per Core) . . . 26
2.5 MCAL Distribution . . . 26
2.5.1 Introduction . . . 26
2.5.2 Assumptions of Use . . . 27
2.5.3 Constraints . . . 28
2.5.4 Definition of MCAL Users . . . 28
2.5.5 Multiple Partitions versus Multi-Core MCAL . . . 29
2.5.6 Multi-Core Capabilities Classification Criteria . . . 30
2.5.7 Definition of MCAL Multi-Core Types . . . 31
2.5.8 Mapping MCAL Modules to Multi-Core Types . . . 35
2.5.9 Separation Strategies and Mapping of Elements . . . 37
2.5.10 Separation Strategies . . . 37
2.5.11 Mapping of Elements . . . 40
2.5.12 Examples . . . 44
2.6 Mapping Software to different Core Partitions . . . 45
2.6.1 Allocation with Global scope . . . 45
2.6.2 Allocation with Local scope . . . 46
2.6.3 Allocation using Cloning capabilities . . . 47
2.6.4 How to determine the Core Scope? . . . 48
2.7 Com-Stack Distribution . . . 52
2.7.1 Introduction . . . 52

2.7.2 Assumptions of Use . . . 53
2.7.3 Constraints . . . 53
2.7.4 Functional Elements . . . 54
2.7.5 Architectural Components . . . 57
2.8 Crypto-Stack Distribution . . . 61
2.8.1 Freshness value handling . . . 61
2.9 Multi-Partition Service Component Architecture Pattern . . . 62
2.9.1 Introduction . . . 62
2.9.2 Model Aspect . . . 63
2.9.3 Implementation Aspect . . . 64
3 BSW Distribution in Safety Systems 66
3.1 General overview on safety . . . 66
3.2 Safety solutions in AUTOSAR . . . 66
3.2.1 Some modules are always ASIL . . . 69
3.2.2 Overall configuration . . . 69
3.2.3 Crossing partition boundaries . . . 70
3.2.4 Access to peripherals / hardware . . . 79
3.2.5 Startup, Shutdown and Sleep/Wakeup . . . 81
3.2.6 Error handling . . . 82
3.2.7 Timing protection . . . 82
3.2.8 Combining Safety and Multi-Core . . . 83
3.2.9 Performance Considerations . . . 83
3.2.10 Constraints . . . 84
4 Outlook on Upcoming AUTOSAR Versions 86
4.1 Known limitations . . . 86
4.2 Inter BSW module calls in distributed BSW . . . 86
4.3 Standardized BSW functional clusters . . . 86
5 Glossary 88
5.1 Acronyms and abbreviations . . . 88
5.2 Technical Terms . . . 88
6 References 90

1 Introduction

This document is a general introduction to the distribution of BSW in AUTOSAR systems. It consists of two parts, one focusing on the distribution of BSW in case of multi-core and the other focusing on distribution in case of safety.

Chapter 2 guides to the development and configuration of AUTOSAR-compliant software for multi-core systems. As of release 4.1, it addresses the allocation of AUTOSAR BSW modules to partitions on multi-core systems and their interaction only. The allocation of BSW modules to different BSW partitions allows for both enhanced functional safety and increased performance.

In Chapter 3 the BSW distribution in safety cases is described. As of release 4.2 AUTOSAR allows to map BSW modules into different partitions and to protect those partitions against each other.

Chapter 4 gives an outlook of possible future extensions in the area of BSW distribution.

A glossary of technical terms and a list of references to external information are provided in Chapter 5 and Chapter 6.

2 BSW Distribution in Multi-Core Systems

2.1 Overview

This chapter contains a description of the supported scenarios for distributed execution of BSW modules on several partitions and cores and a number of use cases in which a distribution of the BSW can enhance performance. It also introduces basic synchronization concepts applicable to distributed BSW execution, and an introduction to inter-partition communication.

2.1.1 Supported Scenarios

It is possible to assign functional clusters of BSW modules ("BSW Functional cluster"), which are used by applications to access buses, non-volatile memory, I/O channels, and watchdogs, to different BSW partitions for safety or performance reasons. The clustering of BSW modules is currently not standardized. Except for the MCAL, parallel usage of the same type of functional clusters in different partitions ("duplication") is not generally supported, but it is possible by using a master satellite approach. Functional clusters to partitions may be assigned such that

- a BSW functional cluster is only available in one partition
- a BSW functional cluster is available on all partitions with all interfaces
- a BSW functional cluster is distributed over multiple partitions, possibly with partition specific subsets of functionality, to allow a high grade of concurrency.

By supporting the scenarios listed above, AUTOSAR addresses the following essential features:

- All code for communication between BSW partitions can be generated for automatic adaptation to different system configurations. The cross partition communication mechanism can be generated with focus on efficiency, or, in future releases to help to provide freedom from interference.
- If access to system services (which are not part of a BSW functional cluster) is required, the according interfaces shall be provided to each BSW partition that needs the system service.
- Efficient access to HW abstraction and drivers is supported in each BSW partition.

In all scenarios, the communication between different module entities remains unchanged (in comparison to BSW running in a single partition).

2.1.2 Performance Use Cases and Hardware Assigned to Different Cores

The following use cases are examples for how system performance can be improved by allocation of the BSW to multiple partitions and cores, and how systems where the access to the peripheral hardware is assigned to multiple cores benefit from the allocation of the BSW to multiple partitions and cores.

- To increase system performance and to reduce resource consumption in systems that are distributed over several cores, it may be necessary to allocate functional clusters of BSW modules to different cores, e.g. communication modules on BSW partition "A" and I/O modules on BSW partition "B", depending on hardware architecture, load balancing and on distribution of SW-Cs. In particular, if HW resources are accessed exclusively by one core in a Multi-Core system, the performance is increased by locating the corresponding BSW users, services and drivers on that core.
- Signal gateway functionality is implemented by allocating a FlexRay cluster on one core and a CAN cluster on a different core. The two COM modules need to be synchronized in this case, and there must be some direct cross core communication between the two COM instances. One of the COM modules might be the master COM that coordinates the satellite COM on the other core.
- Two communication clusters are located on different cores, one accessing a CAN bus and the other one controlling a FlexRay bus. In case the application SW located above one of the communication clusters on the same core needs to send on both buses, the core local COM modules can directly communicate with their counterparts on the other core, to efficiently send the signal over either CAN or FlexRay. For received messages, COM has no information about receivers above the RTE. Therefore, COM has to forward the signals on the receiving side to the RTE, and the RTE is responsible for communication.

2.1.3 Technical Overview

Below is a short summary of the technical solution as described in the following sections:

- Define clusters of BSW modules that contain preferably all three layers of a stack, or, if needed, a subset of modules of a stack (e.g. communication, memory, I/O stack).
- Some BSW modules are widely used by different clusters (and/or BSW modules). For these BSW modules it is recommended that they are implemented in a way that they support callers from different clusters/partitions/cores. Examples of such modules can be found in different areas of Classic AUTOSAR: error handling (Det, Dem), intrusion detection (IdsM), routing (PduR, LSduR), crypto (Csm, KeyM), NV handling (NvM) or Time Synchronization (StbM). A BSW module implementation shall provide sufficient information about possible restrictions for callers from different partitions/cores.

- A module initialization may be split between several partitions e.g. for setting up hardware which takes more time and shall not block the rest of the system.
- Module entities can be split into a master and satellites, which are assigned to different BSW partitions. Masters and satellites can use non-standardized AUTOSAR interfaces, for internal cross partition communication. The master/satellite approach is mainly used by distributed system service modules and for communication between BSW clusters of the same type.

The proposed solution meets the demands on performance and safety while minimizing the impact on already standardized BSW module interfaces. Most changes are hidden within modules (e.g. by providing master/satellite implementations) without affecting other modules. Interfaces between different modules do not change.

2.1.3.1 BSW Functional Clusters

BSW functional clusters are groups of functionally coherent BSW modules. Each functional cluster includes a set of BSW modules. It is possible to have several BSW functional clusters of the same type (e.g. several I/O clusters in different BSW partitions), each using a different set of modules (e.g. IOHWA + ADC in one partition and IOHWA + ADC + DIO in the second partition).

The following types of clusters might be standardized in a later release:

- Communication cluster
- Memory cluster
- I/O cluster
- Watchdog cluster

The allocation of BSW functional clusters to BSW partitions is determined by the usage of BSW modules by the application software. Functional clusters can be allocated to different BSW partitions, and functional clusters of the same type can be available in several BSW partitions. Different functional clusters can be allocated to the same or to different BSW partitions.

The same functional cluster can only exist at most once per BSW partition.

BSW functional clusters are used by applications or other BSW modules to access buses, memory, I/O channels and watchdogs, and they are usually required in one or few BSW partitions only.

The introduction of BSW functional clusters does not change the existing AUTOSAR interfaces between the BSW and the RTE, which are mainly used to implement AUTOSAR services, i.e. to communicate with the application layer. It may however change the availability of standardized AUTOSAR interfaces on different partitions.

The internal structure of a BSW functional cluster, including its internal communication between BSW modules, and the communication with system services that the BSW

functional cluster uses is not necessarily affected by the parallelization of the BSW, and it does not need to change. It may however be adapted, for example in order to fulfill special demands on concurrency like the support of different entities of the same module running in different partitions.

The communication and synchronization between modules in BSW functional clusters of the same type (e.g. in two communication clusters to support a gateway functionality) is not standardized. It will be implemented by communication between entities (e.g. by a master and satellites) of specific modules, which can use non-standardized interfaces for communication across BSW partition boundaries, see Figure 2.1.

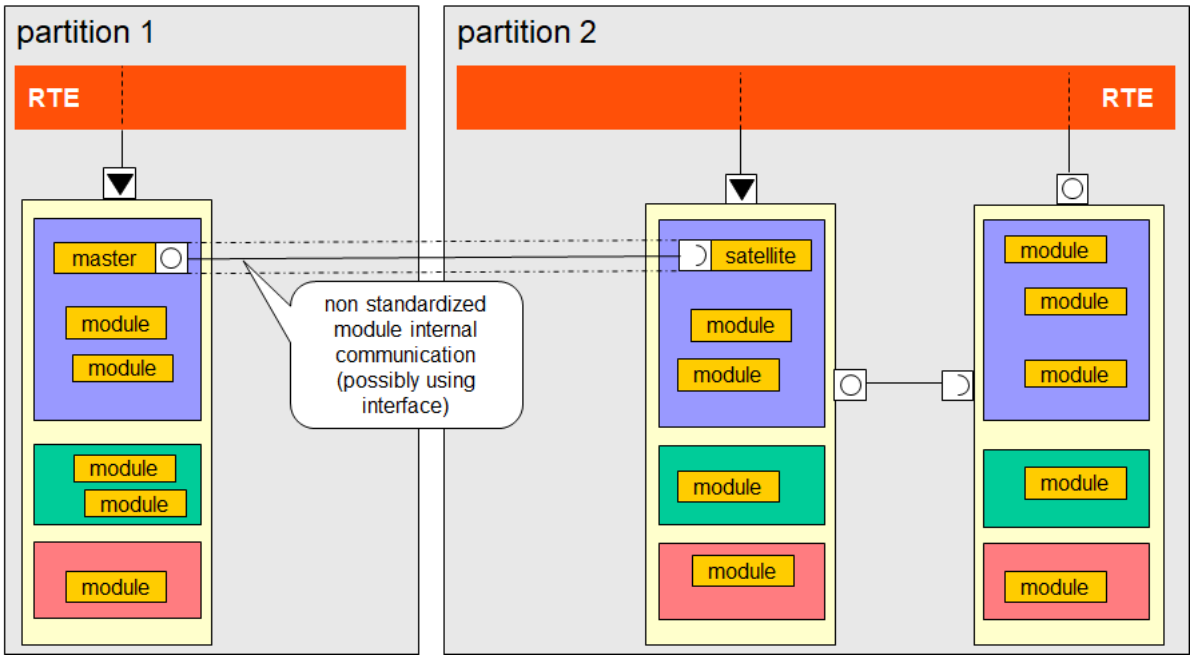


Figure 2.1: Functional clusters of the same type

Modules that do not belong to BSW functional clusters (e.g. system services) will always be accessed within the same BSW partition where the BSW functional cluster is located. As the interfaces do not change, these modules must be locally available in each BSW partition, if needed.

2.1.3.2 Inter-BSW-partition communication

Function calls to tasks that are supposed to be executed in a different BSW partition/ on a different core cannot be implemented as simple C calls to this function, because these calls would be handled on the local BSW partition.

The BSW Scheduler (SchM) therefore provides functions to invoke masters or satellites of the same module on different BSW partitions using either client-server or sender-receiver communication. Details on this API of the SchM are explained in Chapter 2.2.3.

2.1.3.3 Determining the Partition for Service Execution

The actual BSW partition for the handling of an RTE event is determined by its task mapping. Basically, if an event is mapped to a task, it is executed within the partition assigned to this task. If an event is not mapped to a task, it is executed within the same partition as the task that caused the event. Details on the task mapping are described in Chapter 2.4.1 of this document.

Calls from BSW entities to other BSW entities are not mapped to a partition. They are executed wherever they are called. Therefore, several calls to a BSW function may be processed in parallel on different partitions and cores. Consequently such functions must be designed and implemented carefully w.r.t. parallel execution in different partitions; if necessary, they shall be reentrant or concurrency safe.

2.2 Parallel Execution of BSW modules

This is the chapter for developers of BSW modules.

2.2.1 Core-Dependent Branching

Because entities of the same module share the same implementation, even if they are running on different cores, different behavior cannot be realized by different code. Instead, the specific behavior shall be determined by runtime information. It is possible for example to use the core id for this, i.e. branch the control flow depending on the return value of the OS APIs `GetCoreID`, or also `GetApplicationID`.

Another variant of implementing modules operating sharing the same implementation but running on different cores can be realized basing on a different core individual configuration. This requires to call the initialization routine `Init` per core passing a pointer to the according configuration. This design pattern is considered as ideal to implement a core-dependent branching for the MCAL.

2.2.2 Master/Satellite-approach

Modules that need to be accessed in different BSW partitions can be implemented using the master/satellite pattern.

The distribution of work between master and satellite is implementation specific. One extreme is that the satellite only provides the interfaces to the other modules in the same BSW partition, and that it routes all requests to the master and answers back to the other modules. At the other extreme, the satellite can provide the full functionality locally (e.g. local mode management for a complete application which runs in the same BSW partition) and only synchronizes its internal states with the master, if necessary. There might even be several masters for different functionality, e.g. two PduR masters for a distributed PduR gateway.

The master coordinates requests from the satellites and can filter or monitor incoming satellite requests. The master and one or several satellites are treated like being one module entity in some respect:

- Master and satellites are always vendor specific solutions, coming from the same vendor.
- The interfaces of master and satellite to other module entities in general are the same as specified in AUTOSAR for traditional modules. Master and satellite should provide the same APIs. This means that when migrating to partitioned systems, existing module entities can be replaced by a master and one or several satellites, in most cases without changing other modules. Exceptions might be module internal adaptations to additional delays which are caused by inter-partition communication.
- Master and satellites have the same entry points in each BSW partition (i.e. they start executing the same functions from shared memory) and internally branch (e.g. by using the `GetApplicationID` API) to master or satellite specific code according to the OS-Application (partition) they run in. Depending on the build strategy, other implementations might be possible in multi-Core systems if each core can execute its own code. Also, satellites might share the same code without further branching.
- As an alternate realization the master- satellite approach could be implemented in a way that the master is realized as a satellite too, while the real master implementation consists of the BSW module kernel only so that all requests can be exchanged with this kernel. This approach is considered as ideal for MCAL implementations.
- The communication between master and satellites is not standardized. It is considered to be module-internal and is not visible to other modules.
- The communication between master and satellite can be initiated in either direction (i.e. by both the master and the satellites), as well as from one satellite to another one.
- All interfaces between masters and satellites are only allowed to be connected within the same distributed module.
- The communication between master and satellites can be implemented within one `BswModuleEntity`, or between different `BswModuleEntity`s that belong to the same BSW module.
- Depending on the application, usage of master/satellite may be appropriate or not. For example, it may be more efficient to use separate, partition specific watchdog clusters, which work independently from each other, rather than using the Watchdog Manager in a master/satellite approach.
- The master is the part of a distributed BSW module that coordinates requests by satellites and can filter or monitor incoming satellite requests. This may result

in additional fault detection or fault mitigation mechanisms. Generally, all errors caused by distributed execution of a module should be handled module internally.

The master/satellite implementation is the standard solution for system services in partitioned systems.

Specific drivers also might have to provide local satellites, if the hardware can only be accessed from a different core. The standard solution, if possible, is to execute the same multi-core reentrant function in each partition and to separate the data to work on into disjoint sets, one for each partition. For example, the COM module may work on all IPDUs assigned to the bus that the BSW functional cluster of this module belongs to. Concurrent access to the same hardware or shared data needs to be protected, e.g. by ExclusiveAreas in this case.

In specific cases, modules within BSW functional clusters also need to be implemented as master/satellite, if the BSW functional clusters are duplicated and the entities in different BSW partitions need to be synchronized or need to exchange data. This might apply to the Watchdog Manager, the NVRAM manager, and to network and state managers in duplicated communication clusters. COM modules also might need to have a master and a satellite to implement cross partition gateway functionality.

2.2.3 Using the BSW Scheduler for Inter-Partition-Communication

The BSW Scheduler (SchM) provides a number of functions to support communication between BSW module entities that are executed in parallel. More precisely, it provides the following methods to handle synchronous and asynchronous calls (including callbacks) as well as sender-receiver communication.

The functionality is generally similar to that of function calls between SWCs and the BSW. However, because the RTE may not be available at certain points of time (especially during startup of an ECU), this functionality must be available within the BSW itself.

- `Std_ReturnType SchM_Call_<bsnp>[_<vi>_<ai>]_<name>(`
  `[OUT <typeOfReturnValue> returnValue]`
  `[IN|IN/OUT\|OUT]<data_1> ... [IN|IN/OUT|OUT] <data_n>)`

  or

  `Std_ReturnType SchM_Call_<bsnp>[_<vi>_<ai>]_<name>(`
  `[IN|IN/OUT\|OUT]<data_1> ... [IN|IN/OUT|OUT] <data_n>)`

  Invoke a client-server-operation, possibly crossing partition boundaries. The actual parameters `data_1` ... `data_n` are information that is passed [IN] and/or re-passed [IN/OUT | OUT] to/from the called service.

  The presence of the parameter `returnValue` and its type `<typeOfReturnValue>` depend on the called service. For synchronous calls, the parameter is present and `<typeOfReturnValue>` is the type returned by the called service.

For asynchronous client-server-operations and operations with return type `void`, the parameter is omitted.

- `Std_ReturnType SchM_Result_<bsnp>[_<vi>_<ai>]_<name>( [IN|IN/OUT|OUT]<data_1> ... [IN|IN/OUT|OUT] <data_n>)`

  Callback from an asynchronous client-server-operation, possibly crossing partition boundaries.

  The receiver of a callback is determined by the AsynchronousServerCallResult Point of this callback. The AsynchronousServerCallResultPoint refers to the originating AsynchronousServerCallPoint, which in turn "knows" the calling module entity.

- `Std_ReturnType SchM_Send_<bsnp>[_<vi>_<ai>]_<name>(IN <data>)`

  Write data to a sender-receiver link between BSW modules, possibly crossing partition boundaries.

- `Std_ReturnType SchM_Receive_<bsnp>[_<vi>_<ai>]_<name>(OUT <data>)`

  Read data from a sender-receiver link between BSW modules, possibly crossing partition boundaries.

2.2.4 Using Shared Buffers (in systems without memory protection)

In systems without memory protection between the BSW partitions, system services and all BswCalledEntities can be called directly in every partition, including the complete call tree. This requires a reentrant, concurrency safe implementation.

The services and other called entities might work on module internal data, which is shared between different entities of the same module. All access to such data must be protected by ExclusiveAreas. Appropriateness of concrete protection mechanisms depends on the possible kinds of access. For example, concurrent writing generally needs to be prohibited, whereas concurrent reading may be acceptable, as long as only one partition writes at the same time.

BswSchedulableEntities are located on one core only and process the data periodically or event driven.

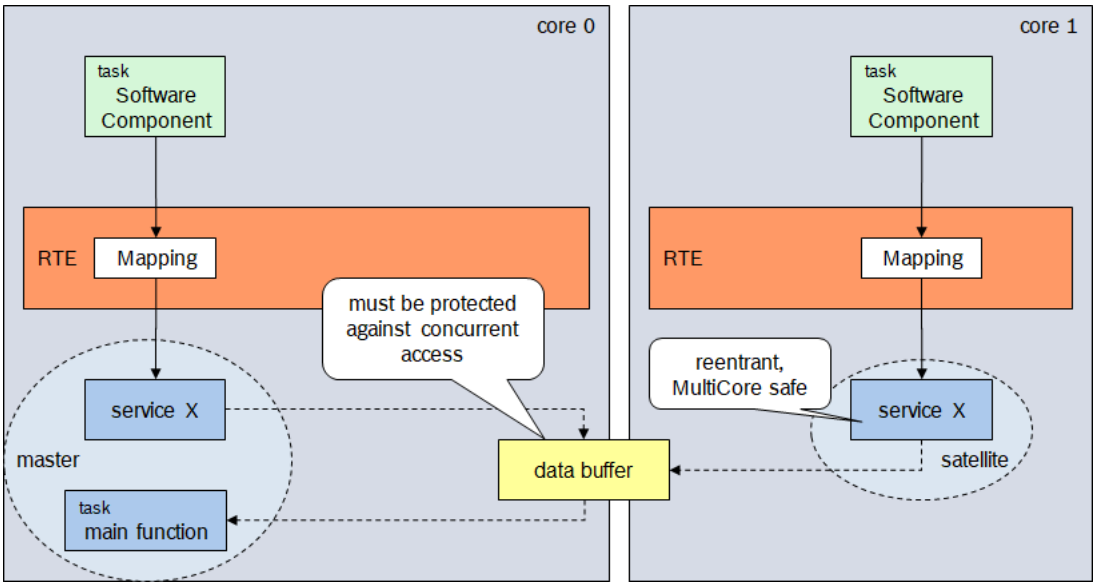


Figure 2.2: Invocation of same service on different cores

Figure 2.2 shows the example of a service "X", where the same API and the same code is called directly by the RTE on different cores. This is the default, if the services (respectively the OperationInvokedEvents) are not mapped to a task.

The code must be reentrant and concurrency safe, which means that all access to data must be protected against concurrent access by the same or by a different entity of the same module.

In this example, the same service "X" (BswCalledEntity) writes into a module internal data buffer accessible from core 0 and from core 1. A "main function" (`BswSchedulableEntity`), which is mapped to a task, reads the data from the buffer for further processing. In order to prevent read/write-conflicts, this "main function" must be protected from reading the buffer while it is written.

This can be considered a special case of the generic master/satellite approach for systems without memory protection between the BSW partitions.

The advantage of this approach is that the original, unchanged modules can be used, as long as they are implemented concurrency safe, which is usually the case for single core already, if different entities of the same module work on the same data, as shown in the example for core 0. Compared to the AUTOSAR R4.0 solution, where all service calls have to be routed to the master core, the performance can be improved considerably without much effort (assuming there is no need to do cross-core communication later).

The following must be considered for a concurrency safe, reentrant implementation:

- Access to all shared resources, e.g. buffers, is protected by ExclusiveAreas.
- Call trees can be made multi-core safe, if either called entities are safe, or calls are protected by ExclusiveAreas (if lock times stay within a specified limit).

BswCalledEntities that are available to CDDs can also be called directly by the CDD. The same rules apply as in R4.0.

The SchM must support cross core ExclusiveAreas, implemented by protected Spinlocks. A protected spinlock is an exclusive area that has "OS_SPINLOCK" as its value of `RteExclusiveAreaImplMechanism`. This kind of exclusive areas is available for controlled access by BSW modules only. Protected spinlocks are handled by the Basic Software Scheduler.

2.2.5 Accessing Hardware/Drivers

`BswModuleEntity`s of the MCAL (drivers) shall be accessed in the following way:

- Access by the BSW functional cluster within the BSW partition where the caller is located. So for example the FLS driver belongs to the BSW functional cluster "Memory". In case of NVM access, the NVM module might be provided on all cores as a master/satellite implementation. The master uses the FLS driver on a single core only. So the FLS driver is available on exactly that core.
- Any BSW required by the application shall be accessed in the BSW partition where the caller is located. For example I/O drivers such as DIO, ADC and PWM can be used by any core / partition. These are either realized as master/satellite implementation or as a redundant implementation per core basing on atomic access to the hardware.

The detailed realization of the MCAL multi-core approach is described in Chapter 2.5 “MCAL Distribution”.

2.2.6 Concurrency safe implementation of modules

Concurrency safety of BSW modules respectively the functions implemented by these modules may be achieved by different mechanisms.

Generally, the following levels of reentrancy can be distinguished according to [TPS_BSWMDT_04103]. The concrete level of a `BswModuleEntity` is defined in the optional attribute "reentrancyLevel".

- **Multi-core reentrant:** Unlimited concurrent execution of an interface is possible, including preemption and parallel execution on multi-core systems. This level can be either achieved by mutual exclusion when entering critical regions, or by the absence of such regions, for example if there are no shared resources (including hardware and memory).
- **Single-core reentrant:** Pseudo-concurrent execution (i.e. preemption) of an interface is possible on single core systems. This is the highest level of reentrancy

defined by AUTOSAR 4.0.3. Because it does not explicitly cover multi-core systems, "concurrency safe" has been introduced additionally. This level can generally be ensured by the same mechanisms as "concurrency safe", but they must be ensured to work across core boundaries.

- **Non-reentrant:** Concurrent execution of this interface is not possible.

If a module that is not concurrency safe is invoked in different partitions, there is no warranty that the module will uphold its desired behavior. In this case, correct behavior shall be ensured by the usage of the module, for example if the caller(s) prevent parallel execution by using exclusive areas.

2.2.7 Kernel based Master-Satellite Realization

One way of realizing the master-satellite concept is the implementation of a module split into kernel and according interfaces, which are provided for all partitions the module shall be used from. The focus of the chapter is to describe the idea of the distribution of the according BSW module as a guidance for similar cases.

In a first step the service API's are categorized, e.g. according to MCAL distribution concept:

- Service API required by multiple cores / partitions for triggering and reading / writing of data
- Service API which are required by one partition only (e.g. initialization, shutdown)

This will lead to the following BSW module architecture:

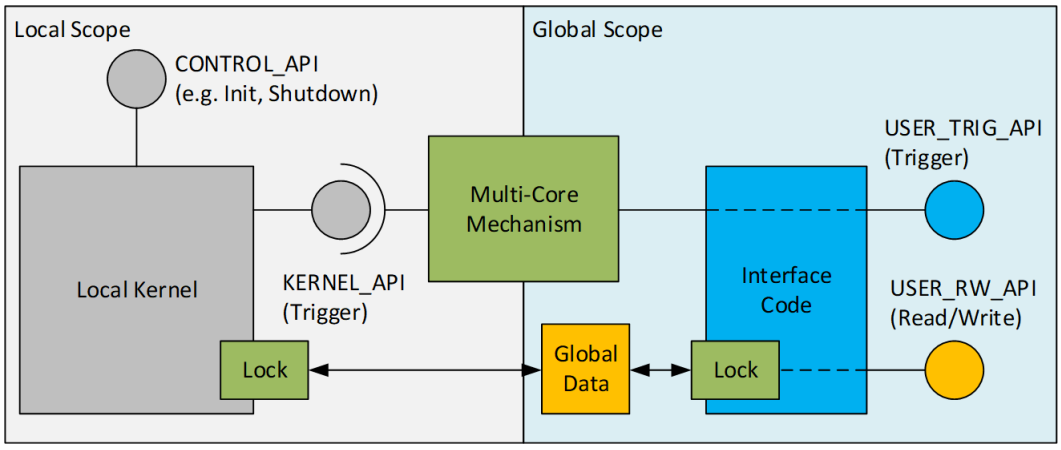


Figure 2.3

In the second step the SW designer needs to select an according BSW distribution pattern for each service API following the below listed categories:

- Trigger a control flow from service API to kernel API (if required including data)
- Read of a symbol from a global buffer

- Write a symbol to a global buffer, which is later-on polled by the kernel

For reading and writing data, one will typically implement according get and set operations into the interface part of the BSW module, adding an according data protection mean (e.g. spinlock) if necessary.

For transferring control flow, one will implement on the one hand side the user API operation and on the other hand the same operation inside the kernel API. The relation can be 1:1 (one user API matches one kernel API) or n:1 (several user API match one kernel API).

All local API have to be allocated with local core scope and all global API with global core scope applying the memory allocation specification (AUTOSAR_SWS_Memory Mapping, see [1]).

Applying this idea to the PWM driver, one would get the following API distribution:

Control API with local scope:

CONTROL API	Core Scope
`Pwm_Init`	local (same partition as kernel)
`Pwm_DeInit`	local (same partition as kernel)
`Pwm_Main_PowerTransitionManager`	local (same partition as kernel)

Note that these API's are restricted to the local kernel partition. The task calling the transition manager needs to be scheduled within the same partition / on the same core.

User read/write API with global scope:

USER_RW API	Access / Core Scope
`Pwm_GetOutputState`	read global data
`Pwm_GetCurrentPowerState`	read global data
`Pwm_GetTargetPowerState`	read global data
`Pwm_DisableNotification`	write to global data (atomic flag)
`Pwm_EnableNotification`	write to global data (atomic flag)

Note that these API's are available to any partition, so these are also used if the call is done from the same partition where the kernel resides.

User trigger API with global scope and related kernel API with local scope:

USER_TRIG_API / Global Scope	KERNEL_API / Local Scope
`Pwm_SetDutyCycle`	`Pwm_Kernel_SetValues`
`Pwm_SetPeriodAndDuty`	
`Pwm_SetOutputToIdle`	`Pwm_Kernel_SetOutputToIdle`
`Pwm_SetPowerState`	`Pwm_Kernel_PowerStateManager`
`Pwm_PreparePowerState`	

Note that the USER_TRIG_API is available to any partition, so these are also used if the call is done from the same partition where the kernel resides. The KERNEL_API however is not accessible to the user. It is internal only. The calls from the local partition are passed through the USER_TRIG_API.

Finally, the picture below shall summarize the interface availability along the cores / partitions for the given example, while the kernel API is hidden inside the BSW module not visible to the outside.

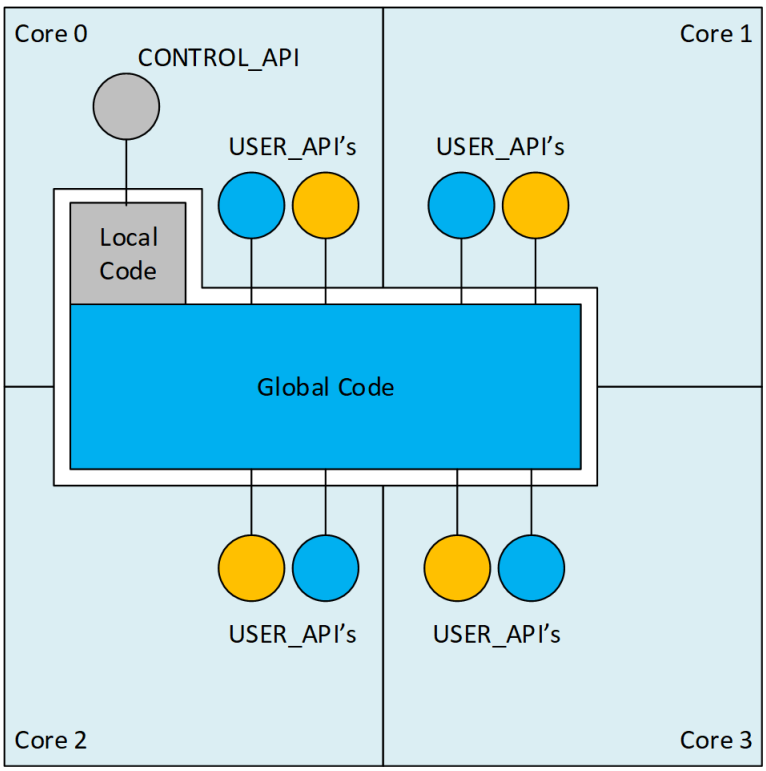


Figure 2.4

The following points shall be noted in addition:

- All user callbacks shall be called in the local partition. The user needs then to provide similarly a service API with global scope, which can be called. This is useful as the implementer of each BSW module or CDD knows ideally how its module is designed. Doing the other way around could waste CPU resources instead as caller and called module might implement the partition transfer twice.
- It is strongly not recommended to apply this or other master-satellite concepts for crossing partitions on the same core, e.g. motivated by safety, as this will also waste CPU resources.

When choosing the according protection and multi-core means one shall always try to achieve a blockade free implementation to allow parallel operation along multiple cores.

2.2.8 Atomic Operations Library

Introducing the BMC Libray, AUTOSAR now provides a multi-core atomic library. The interfaces of this library cover similar use cases / interfaces as the C11 standard atomic library (http://en.cppreference.com/w/c/atomic).

The library shall support developers implementing efficient lock-free implementations and so ease the implementation of distributed BSW (Master/Satellite) without the use of a heavy weight mutex. Please note however that not all HW targets allow lock-free implementation.

Atomic operations typically perform a read-modify-write sequence on a memory address. For example, an atomic increment loads a value, increments it, and stores the result in such a way that no other thread can modify the value in the middle. So, a cross-core set request can be realized with atomic operations in an efficient way.

2.3 SchM Interfaces for Parallel BSW execution

This chapter describes the extensions to the SchM required by the concept "Enhanced BSW allocation".

The Basic Software Scheduler (SchM) is responsible for handling the inter-partition communication between BSW modules. This is conceptually similar to the handling of inter-partition communication between SW-Cs by the RTE. Because the BSW modules are arranged below the RTE in the AUTOSAR architecture however, the communication must be available before the RTE is available. Therefore and for reasons of performance, BSW modules use the SchM for communication.

For the distribution of BSW modules across several partitions, the SchM shall implement the methods `SchM_Call`, `SchM_Result`, `SchM_Send` and `SchM_Receive`, which are used to handle service calls and callbacks as well as writing data to and reading data from a sender-receiver connection. For details on the signatures of these functions, please refer to Chapter 2.2.3, which describes the SchM extensions from a BSW developer's point of view.

The SchM can use `IocSend` (a direct call to the OS) to send data in inter-partition communication. Other RTE internal mechanism might not be available during startup.

The Inter-OS-Application Communicator (IOC) shall be configured to provide `IocSend_<Id>` functions with a uniquely determined `<Id>` for all client-server and sender-receiver connections that cross partition boundaries.

Analogously, the SchM shall use `IocReceive` to receive data from inter-partition communication, and the IOC shall provide the corresponding `IocReceive_<Id>` functions.

The following frame contains some pseudo code snippets that show how to use the IOC for inter-partition communication.

```
1 void some_BSW_function() {
```

```
2     char *str = "some text";
3     SchM_Send_Data_Src_DstN(str);
4   }
5
6   Std_ReturnType SchM_Send_Data_Src_DstN(char *str) {
7     IocSend_1(str, 5);
8     ActivateTask(TASK1);
9   }
10
11  Std_ReturnType SchM_Receive_Data_Src_DstN(char *str) {
12    IocReceive_1(str);
13  }
14
15  TASK(TASK1) {
16    char data[20];
17    SchM_Receive_Data_Master_Sat1(data);
18
19    /* do something with data */
20  }
```

2.4 Configuration of Basic Software in Partitioned Systems

This is the chapter for integrators.

2.4.1 Task Mapping

The parallelization of BSW modules introduces several new subclasses of `BswEvent` to the AUTOSAR metamodel. These classes are shown in Figure 2.5. Each `BswEvent` (including instances of subclasses of `BswEvent`) is assigned to a `BswSchedulable-Entity`, which is started upon occurrence of the event.

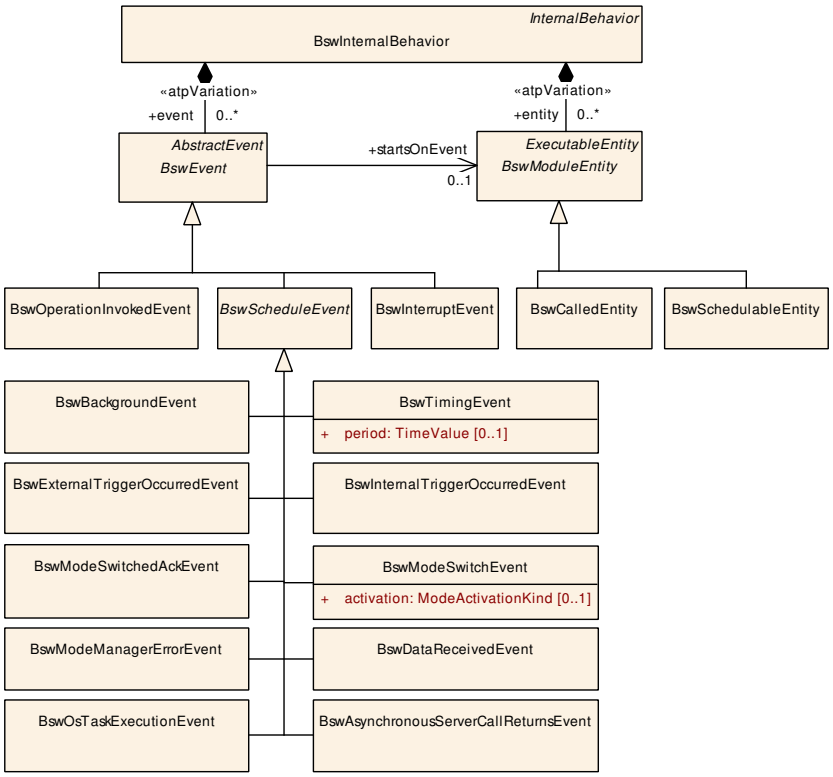


Figure 2.5: Events triggered by the invocation of BSW functions

A more fine grained description of the partition specific behavior of an entity can be described by the use of `BswDistinguishedPartition`s, as shown in Figure 2.6. A `BswDistinguishedPartition` is the abstract representation of a partition, which allows to the mapping of a specific `BswEvent`, `BswModuleCallPoint` or `BswVariableAccess` to a set of abstract partitions. The representation of a partition at this point is an abstract one in the sense that it is part of the BSW module description (according to the module description template), whereas a concrete partition is determined at ECU configuration time.

For example, if a module entity running in partition 1 provides data via a VariableData Prototype to the same entity running in partitions 2 and 3, the `BswModuleEntity` aggregates a dataSendPoint with a contextLimitiation to partition 1 and a dataSend Point with a contextLimitation to partitions 2 and 3.

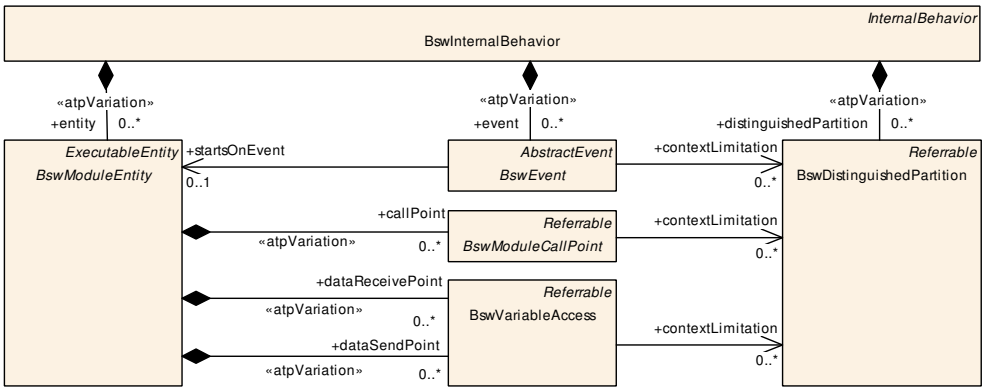


Figure 2.6: Modeling partition specific properties of entities using `BswDistinguishedPartitions`

The actual partition for the handling of an event is determined by its task mapping.

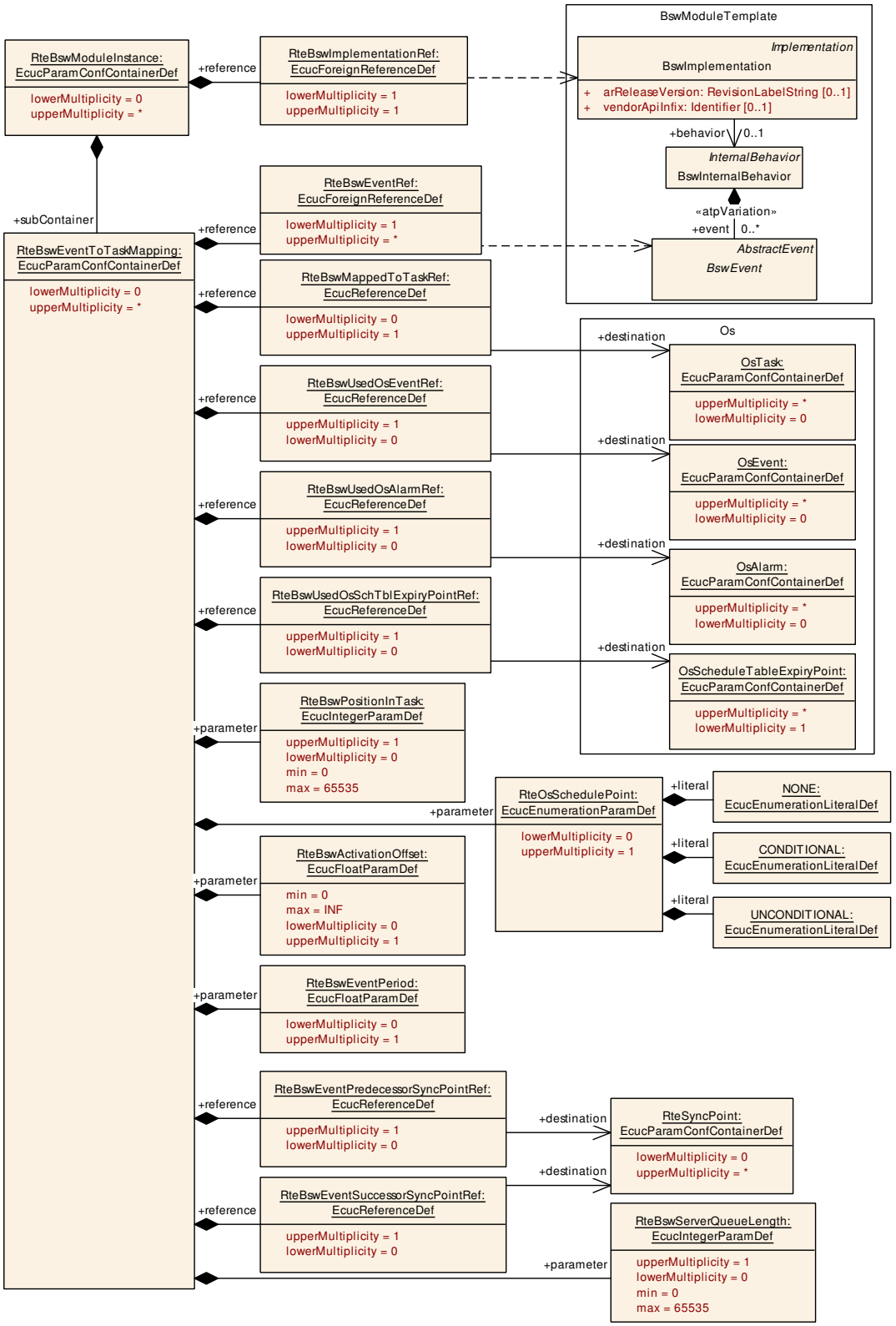


Figure 2.7: Mapping OperationInvokedEvents to tasks

Figure 2.7 shows the corresponding excerpt from the AUTOSAR metamodel.

An RteBswEventToTaskMapping refers to a BswEvent (indirectly via its RteBswEventRef) and to an OsTask (also indirectly via its RteBswMappedToTaskRef). The task is in turn mapped to a partition, and the partition is mapped to a μC core, which is the core responsible for the processing of the event. Mapping an event to a task is optional; if an event is not mapped to a task, it is handled in its originating partition. If no special mechanisms apply that prevent concurrent execution, a prerequisite for a non-mandatory mapping of an event to a task is:

- if the BSW entity is shared between multiple BSW partitions the entity needs to be *concurrency safe*
- in case it is exclusively available only on one BSW partition it needs to be at least *reentrant*.

Please note that it is currently not allowed to map RunnableEntities of a SW component to multiple partitions [SWS_Rte_07347]. For BSW it is possible to map the same module entities to different tasks and partitions by using different BswEvents referring to the same entity

2.4.2 General Configuration of Master and Satellites

Modules that shall be available in multiple partitions can be implemented as masters and satellites. In this case, the master and all satellites of the same module share the same code (which may implement core-dependent behavior however) and the same configuration. Hence, a master and its satellites are treated as one module entity w.r.t. their configuration.

The communication between master and satellites is not to be standardized. It is considered to be module-internal and it is not visible to other modules. However, since it is recommended to use SchM mechanisms for internal communication, the non-standardized client-server entries and data accesses in the BSWMD to connect master and satellite need to be configured.

2.4.3 Configuring the BswM (per Partition)

On systems with distributed BSW there is one BSW Mode Manager (BswM) per partition (but one OS and EcuM per core, which is the same as long as we have one BSW partition per core). Each of these BswMs can be configured independently. A BswM mainly interacts with the state managers (ECU state manager and bus state managers, for instance) on the same partition.

The BswM is also responsible for the initialization and shutdown of BSW modules running in the same partition. Therefore, its configuration depends on the mapping of BSW modules to partitions.

The configuration of the BswMs is split across the container `BswMGeneral`, which contains shared configuration parameters of all BswM entities and `BswMConfig` containers, where one `BswMConfig` is defined for each BswM entity. Consequenty, the mapping of a BswM to its partition is defined in the corresponding `BswMConfig` container, which has a `BswMPartitionRef` pointing to the respective partition. This mapping of BswM configurations to partitions ensures that for every partition the correct configuration of the BswM can be determined.

`RteInitializationRunnableBatch`es need to be defined per partition (needing an initialization). Call context is the `BswM_MainFunction` instance of the corresponding partition. In a multi-core system, a synchronization point of the BswM instances is needed to call the `Rte_StartTiming`.

2.4.4 Configuring the EcuM (per Core)

On systems with distributed BSW there is one EcuM per core (even if there are multiple BSW partitions on that core). In other words, on every core there shall be one and only one partition that runs the EcuM. The partition running the EcuM is determined by the `EcuMFlexEcucPartitionRef`, which is specified in the container `EcuMFlexUserConfig` of the EcuM configuration.

On architectures with a sequential start of cores, there is one designated master core in which the boot loader starts the master EcuM via `EcuM_Init`. The EcuM in the master core starts some drivers, determines the Post Build configuration and starts all remaining cores with all their satellite EcuMs.

On architectures where all cores are started at the same time, core dependent branching within the `EcuM_Init` function can be used to achieve core-specific behavior. This can in turn be used to identify the EcuM master (running on the master core), which is responsible for the EcuM initialization on the slaves.

2.5 MCAL Distribution

2.5.1 Introduction

Because it is required to provide access to hardware features from several cores and partitions the MCAL functionality needs to be provided to exactly that core it is required and where it is useful to provide the functionality. So consequently the distribution of MCAL modules is not identically done for all MCAL modules but needs to follow the needs of the functional clusters described in the chapters before. The following chapters shall guide through the classification of the required multi-core capabilities, introduce an according multi-core type which is assigned to the individual modules. Furthermore some basic design patterns shall be shown to allow the implementation of the required functionality.

It shall be noted that the introduction of the multi-core MCAL requires the introduction of asynchronously behaving interfaces to enable non-blocking parallel execution on multiple cores. These are introduced to the individual SWS of the affected AUTOSAR modules and not mentioned furthermore in the chapter below.

2.5.2 Assumptions of Use

To apply the MCAL distribution several assumptions of use shall be given, to define the boundary conditions of the MCAL environment:

1. A multi-partition (multi-application) AUTOSAR operating system is required to support the use cases defined within this concept.
2. The hardware implementation shall allow a mapping of peripherals at least to cores. In future it is expected that hardware implementations allow a mapping to cores and partitions.
3. It shall be possible to route hardware and software interrupts to one partition or at least a dedicated core (for further routing by the OS).
4. Service modules which are required by the MCAL drivers shall support multi-core use-cases by being able to accept calls to their service API's on respectively by any core. The relevant services are:
    - Det
    - Dem
    - EcuM
    - Os
    - SchM
    - NvM

Furthermore it is assumed that a multi-core microcontroller is used however this is not mandatory as the concept provides an identical set of service API's regardless whether it is a single- or multi-core implementation. Additionally it is possible to realize mixed ASIL systems with segregation in space and time where the mappable MCAL elements are assigned to the different partitions respecting the safety integrity level of the resulting MCAL implementation.

An example is a system with two partitions on one core which both access the MCAL. Without this concept, the driver must belong exclusively to one of the partitions, making partition crossings execution time expensive. With the new concept, MCAL elements can be individually assigned to the two partitions and thus eliminating the need to cross partition boundaries.

2.5.3 Constraints

To realize the concept further constraints are defined to prevent inefficient and multi-core blocking implementations. In this sense it is especially important to consider that it is not sufficient anymore to implement exclusive areas on a single core but to additionally ensure an access serialization in case resources need to be shared across several partitions distributed to several cores.

- Access Serialization on a single core: For single-core systems concurrency problems are well understood and mitigated by exclusive areas, which limit concurrent access to one process at a time. This is typically done by locking interrupts, employing OS resources or creating a non-pre-emptive scheduling. This effectively means access-serialization of the different processes.
- Access Serialization across cores: Since exclusive areas only have a core-wide scope, they are not sufficient to prevent concurrent access in multi-core environments. But as soon as it is required to access the same resource (e.g. by access to the service API's, processing of the ISRs and main-functions) it is required to introduce cross-core means. Besides the usage of atomic resources, the worst - because blocking - one would be the introduction of a cross-core exclusive area by using a semaphore (spin lock) which would block several cores. Instead a better option would be a classical master-satellite implementation basing on a proprietary - lean - IOC.
- Asynchronous DMA copy operations of PDUs shall only be used between buffers of modules assigned to the same partition.

As a summary, exclusive areas could technically be extended for multi-core scope however these would be implemented, but these would cause a significant performance drawback as two or even multiple cores would be blocked. So the concept will describe according design patterns showing the optimal protection means aligned to the defined multi-core types within this chapter.

2.5.4 Definition of MCAL Users

There are the following different MCAL users to be considered:

- Application SWC (above the RTE) via IoHwAbstr
- CDD or BSW Module (below the RTE)

So the MCAL multi-core support needs to be provided independently of the RTE to cover both the use-cases.

2.5.5 Multiple Partitions versus Multi-Core MCAL

2.5.5.1 Considering Multiple Partitions

As the AUTOSAR standard allows the definition of multiple partitions per core, the MCAL distribution concept needs to respect this idea for the purpose of later extensions motivated for example by safety distribution. So, this concept implements the following idea:

1. Mappable elements are not simply mapped to cores, but instead to partitions. This is to allow, that one is able to implement further isolation features to the MCAL at a later point in time to ensure for example freedom from interference in between ADC channels.
2. MCAL interfaces instead are not mapped to partitions, but it is assumed that these are available on the cores where the user partitions reside. This results in the correct definition of the core scope, which is either GLOBAL or LOCAL. Simply spoken this means that interfaces required by partitions on a single core only, might be allocated with LOCAL scope. Instead interfaces required by different partitions on multiple cores shall be allocated with GLOBAL scope.

Conclusion: APIs are not mapped to partitions (just related to partitions). Mappable elements are mapped to partitions for further distribution needs.

2.5.5.2 Impact on MCAL Symbol Allocation

Respecting the idea of multi-core on the one and multiple partitions on the other hand will cause some influence on the mapping of MCAL driver internal data, constants as well as peripherals. Depending on the driver design as well as the hardware capabilities one can:

1. Map peripherals to cores or partitions, which might be protected by further hardware means, such as: privilege levels (hypervisor, supervisor, user, ...), safety partitions (by partition ID, task ID, ...) etc.
2. If required, allocate data in the individual partitions to ensure for example freedom from interference in between those. But allocate the data at least with the same core scope than the APIs using it. Map data and related operations to the same core.

Please note that allocating data to different partitions does not isolate those automatically. To achieve this an according memory protection is required in addition.

Conclusion: Symbols shall be allocated with the same core scope than the APIs (internal or public) using it.

2.5.6 Multi-Core Capabilities Classification Criteria

The following paragraphs are given to unify the understanding the required multi-core capabilities from different point of perspective.

2.5.6.1 Criteria 1 - APIs Availability

To classify the multi-core capability of a MCAL module it is first essential to understand the user expectation in the sense of "from what core the service API's shall be reachable". Out of this definition the following two cases can be derived:

- 1a: Local service API's (executable on one core only)
- 1b: Global (distributed /shared) service API's (executable on any core)

2.5.6.2 Criteria 2 - MCAL Kernel Execution Context

Secondly one needs to understand where the MCAL module kernel shall ideally reside/ located to limit the side effects of collisions on busses and bridges due to concurrent access to HW peripherals from several cores. Defining a local kernel does not exclude a multiplicity, to e.g. provide several kernels dealing with independent peripheral modules or core individual resources. The following cases are defined:

- 2a: One Local kernel (executable on one core only)
- 2b: Global (distributed /shared) kernel (executable on any core)

2.5.6.3 Criteria 3 - HW Elements Mapping

As a third point one needs to consider the scope of mappable elements, refer to Chapter 2.5.11, including its data to the according kernel instance. Taking this aspect into account, one extends the classification according to the hardware capabilities in terms of mapping of HW peripherals to cores. Here not only the pure hardware capability needs to be considered but also the performance impact of the according mapping. The following cases are defined:

- 3a: One HW element mappable to one core only
- 3b: One HW element mappable to several cores

2.5.6.4 Multi-Core Capabilities Classification Summary

The following table summarizes the scope of the required options for the shown criteria.

	One Core Only	Several Cores
APIs	1a	1b
Kernel Execution Context	2a	2b
HW Elements	3a	3b

Table 2.1: MC Capabilities Criteria

2.5.7 Definition of MCAL Multi-Core Types

The following paragraphs introduce according multi-core types to be applied to MCAL modules classifying the according multi-core capabilities.

2.5.7.1 MCAL Multi-Core Module Type I

The MCAL modules are available on a single core only, the interfaces are not globally available.

Type I = 1a + 2a + 3a

The type is defined as a single-core module providing its service API's to one core only and implementing the kernel on exactly this core as the according HW elements shall be accessed by one core only.

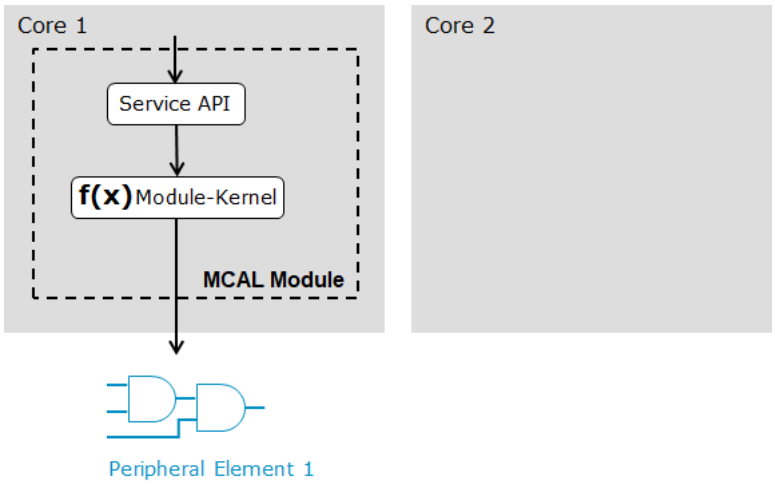


Figure 2.8: Type I

Examples of Type I are FLS, MEMIF and FEE. To limit the scope to this core, an according `SwAddrMethod` with local scope can be applied.

2.5.7.2 MCAL Multi-Core Module Type II

The MCAL modules provides a distributed kernel, executed per core, acting on individually mapped HW elements.

Type II = 1b + 2b + 3a

The type is defined as a special kind of a multi-core module providing its service API's as well as control API's (Init, DeInit etc.) on any core individual instance. So the action is performed on the core the action is triggered on. Each core instance operates on its own set of data. This especially makes sense for MCAL modules operating on HW elements which can be mapped to one dedicated core. A typical example for this type is, communication drivers such as CAN, ETH and FR.

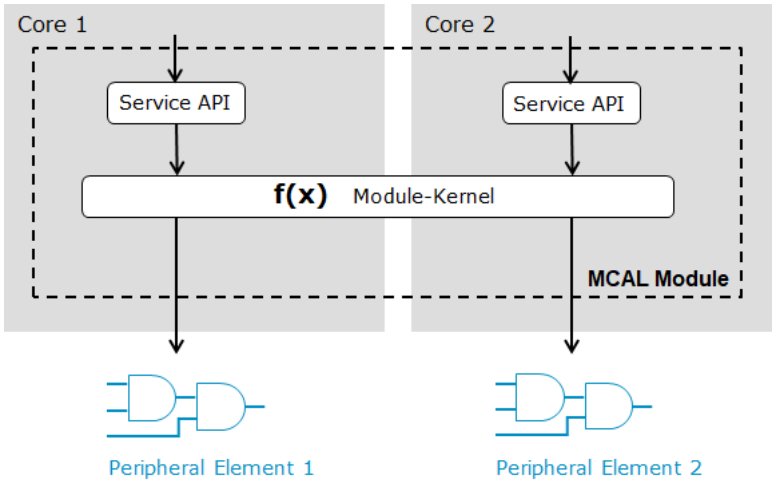


Figure 2.9: Type II

2.5.7.3 MCAL Multi-Core Module Type III

The MCAL modules provides a distributed kernel executed per core acting on globally available HW elements.

Type III = 1b + 2b + 3b

The type is defined as a special kind of a multi-core module providing its service API's on all cores but implementing the kernel in a global manner so that the action is performed on the core the action is triggered on directly accessing the globally available HW elements, mappable to any core including the related data. The according control API's (Init, DeInit etc.) instead are available on one single core only. Especially in case the HW can be accessed atomically this module type is considered as useful. The most prominent example is the DIO driver.

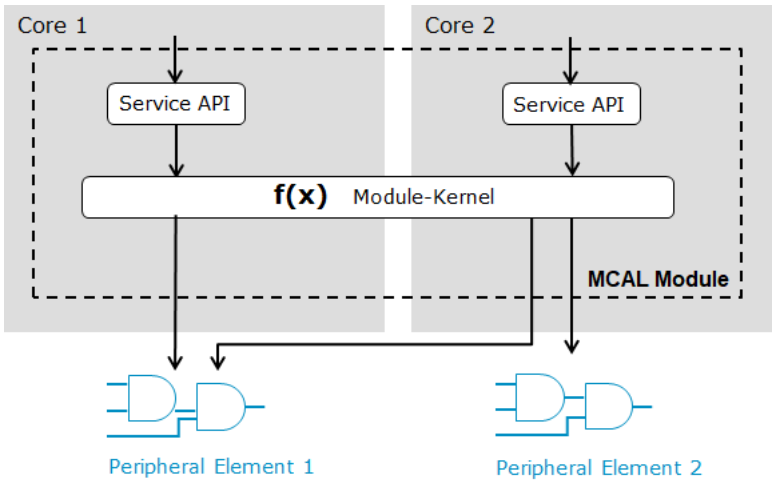


Figure 2.10: Type III

2.5.7.4 MCAL Multi-Core Module Type IV

The MCAL module provides interfaces available on any core and one kernel on a single core accessing the mappable element by one core only

Type IV = 1b + 2a + 3a

The type is defined as a special kind of multi-core module which provides its service API's across all cores but implements the kernel on one core only performing the access the mappable HW elements. The kernel could be allocated with the `SwAddrMethod` "local". This case requires proprietary multi-core means to perform synchronization (serialization) of requests towards the kernel. Such multi-core means could be a highly efficient message passing basing on polling or interrupts, a multi-buffering in combination with semaphores (for low recurrences). The according control API's (Init, DeInit etc.) of the type IV MCAL module are available on the core the kernel resides only with the according local scope. Examples for such a kind of BSW modules are ADC, PWM, ICU and OCU. This is a classical master-satellite implementation.

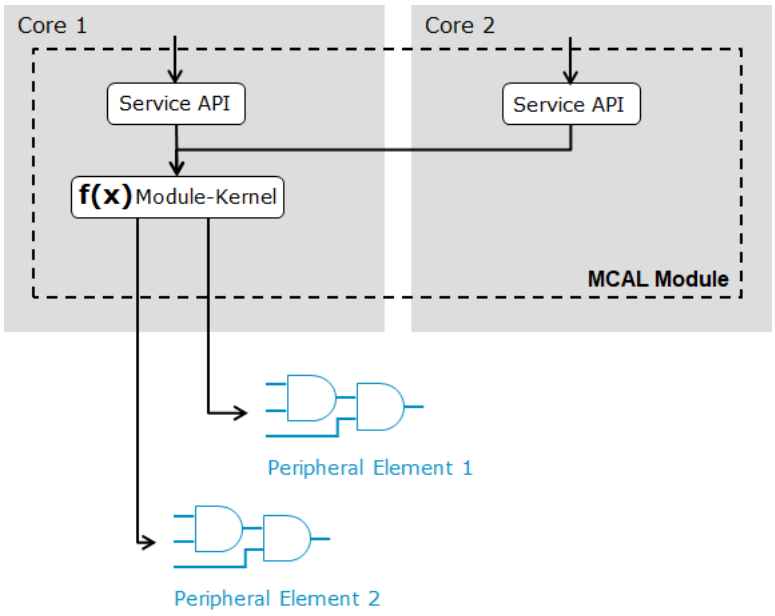


Figure 2.11: Type IV

2.5.7.5 MCAL Multi-Core Module Type V

The MCAL module provides interfaces available on any core and multiple kernels on individual cores accessing the mappable element by the according core individually.

Type V = 1b + 2a + 3b

This multi-core module is an extension of type IV which can be realized with HW implementations which allows a fully independent handling of peripheral modules or sub-modules. This is a rather academic constellation, no example picture is given.

2.5.7.6 MCAL Multi-Core Type Summary

The following table summarizes the scope of the defined MCAL multi-core module types:

	APIs		Kernel Execution		HW Elements	
	Only one Core	Several Cores	Only one Core	Several Cores	Only one Core	Several Cores
	1a	1b	2a	2b	3a	3b
Type I	X		X		X	
Type II		X		X	X	
Type III		X		X		X
Type IV		X	X		X	
Type V		X	X			X

Table 2.2: MC Capabilities Classification

2.5.8 Mapping MCAL Modules to Multi-Core Types

The concept shall be generally applied to all MCAL drivers, which are listed in the following table:

Module Abbreviation	MSN	SW Layer
Adc	ADC Driver	I/O Drivers
Can	CAN Driver	Communication Drivers
CanTrcv	CAN Transceiver Driver	Communication HW Abstraction
CorTst	Core test	Microcontroller Drivers
Dio	DIO Driver	I/O Drivers
Eth	Ethernet Driver	Communication Drivers
EthSwt	Ethernet Switch Driver	Communication HW Abstraction
EthTrcv	Ethernet Transceiver Driver	Communication HW Abstraction
Fr	FlexRay Driver	Communication Drivers
FrTrcv	FlexRay Transceiver Driver	Communication HW Abstraction
Gpt	GPT Driver	Microcontroller Drivers
Icu	ICU Driver	I/O Drivers
Lin	LIN Driver	Communication Drivers
LinTrcv	LIN Transceiver Driver	Communication HW Abstraction
Mcu	MCU Driver	Microcontroller Drivers
Mem	Memory Driver	Memory Drivers
Ocu	OCU Driver	I/O Drivers
Port	Port Driver	I/O Drivers
Pwm	PWM Driver	I/O Drivers
RamTst	RAM Test	Memory Drivers
Spi	SPI Handler Driver	Communication Drivers

Table 2.3: Relevant Modules

To identify the multi-core type and mapping relation of the standardized MCAL modules one first needs to identify the HW "natural element" which shall be accessed by the module. Furthermore one need to identify the mappable element means the element which the user running on an individual core likes to access. Out of the definition one can then derive the relation of mappable elements to cores. Here the mappable element (ME) is shown in relation to the number of cores (Core) it can get mapped to. As a final conclusion the according multi-core type is shown required to derive a later design pattern recommendation for implementing the according AUTOSAR module.

Driver	HW "Natural" Element	Mappable Element (ME)	Relation (ME : Core)	Multi-Core Type
Adc	HW Units	Channel group	n:m	Type IV
Can	CAN Controller	Network	n:1	Type II
CanTrcv	Transceiver ASIC	Network	n:1	Type II
CanXL	CAN XL Controller	Network	n:m	Type III
CanXLTrcv	Transceiver ASIC	Network	n:m	Type III
CorTst	Core	Core	1:1	Type II

△

Crypto	HW based: HSM SW based: Job	Job	n:1	Type II
Dio	Port / Channel (HW dependent)	Port / Channel	n:m	Type III
Eth	MAC	Network	n:1	Type II
EthSwt	Switch ASIC	Network	n:1	Type II
EthTrcv	Transceiver ASIC	Network	n:1	Type II
Eep	EEPROM Driver	MCAL Module	1:1	Type I
Fls	Flash	MCAL Module	1:1	Type I
FlsTst	Flash Test	MCAL Module	1:1	Type I
Fr	Controller	Network	n:1	Type II
FrTrcv	Transceiver ASIC	Network	n:1	Type II
Gpt	Timer Resource	Local Timer Global Timer	n:1 1:m	Type II Type III
Icu	Timer / Edge Detector	ICU Channel	n:m	Type IV
Lin	Lin Channel	Network	n:1	Type II
LinTrcv	Transceiver ASIC	Network	n:1	Type II
Mcu	Core	Core, System	1:1	Type II
Mem	Memory, e.g. Flash	Memory Instance	n:1	Type II
Ocu	Timer	OCU Channel	n:m	Type IV
Port	Port / Channel (HW dependent)	Port / Channel	n:m	Type III
Pwm	Timer	PWM Channel	n:m	Type IV
RamTst	Core	Core, System	n:1	Type II
Spi	Channel (for individual sequences) / Device	Spi Device	n:m	Type IV
Wdg	Watchdog Driver	Watchdog Resource	1:1	Type I

Table 2.4: Relevant Modules

As a conclusion drivers belonging to type I which are consequently not impacted by this concept are listed below. For each driver, a rationale is given why it is deemed to be not relevant.

- Eep (EEPROM Driver): Memory services (NvM) are bound to one core. Hence there is no need for multi-core functionalities of the driver.
- Fls (Flash Driver): Memory services (NvM) are bound to one core. Hence there is no need for multi-core functionalities of the driver.
- FlsTst (Flash Test): Flash Test offers no potential for additional (application) use cases. Its purpose is to Check the functionality of the microcontrollers' flash memory as kind of a service. There are typically no SW functionalities realized with this module.
- Wdg (Watchdog Driver): Although there are multiple watchdogs on a multi-core system, each of them is triggered by just one core locally via a corresponding Task/ISR. The configuration (`Wdg_SetTriggerCondition`) of all the watchdogs is done only on one core to ensure system-wide behaviour.

2.5.9 Separation Strategies and Mapping of Elements

The challenge of the MCAL multi-core distribution is how to deal with global resources. These are:

- Global data
- Shared special function registers
- Peripheral registers

According to the given constraints in the chapters before it is obvious that two process contexts will access an identical global resource simultaneously. This can lead to:

- Corrupted data (Especially a problem with complex (non-atomic) data types.)
  - Part of the data is written by the first process; another part is written by a second process.
  - Only part of the data is written, and then the writing process is pre-empted, leaving a corrupt data-set.
- Races with read-modify-write data:
  - Data written by a process (e.g. increment of a value) gets lost due to two interleaved read-modify-write operations.

In MCAL drivers, there are up to three elements which can have their own process contexts:

- Main function: Mapped- and executed in task context
- Service API: Called in the context of one or several tasks or ISR
- Interrupt Service Routine: Called in interrupt context

Especially service API's might be called in several process contexts. Depending on the architecture and functionality realized by the SW.

This chapter describes the multi-core capabilities according to the mappable element (which corresponds to the functional elements) which are mentioned earlier in this document and which shall be annotated to the MCAL driver. In addition the chapter defines basic separation strategies required to implement the mappable elements.

2.5.10 Separation Strategies

2.5.10.1 Separation on HW Level

One of the ideal ways to be able to realize a multi-core implementation according to the defined multi-core-types is by distribution/separation on hardware level.

Ideal case is when the HW supports distribution/separation of physical peripherals, i.e: mapping peripheral modules to individual cores.

Note: This HW level separation requires independed register-sets of the individual peripheral which can be controlled from one core without impacting another one as shown in Figure 2.12.

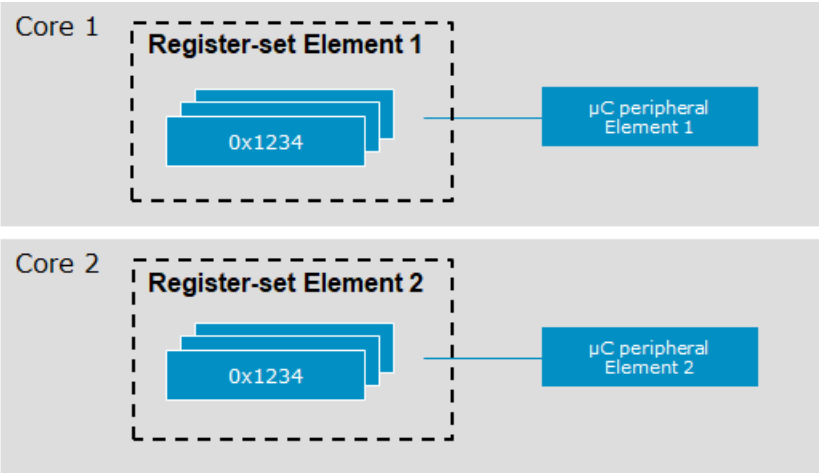


Figure 2.12: Independent Register-Set HW Level Separation

As shown in Figure 2.12, the register-sets behind the individual hardware/peripheral elements are independent of each other and so can be considered as mappable element. Mappable element means, one element can be mapped to a certain core exclusively. In case the register-set element allows an atomic access, mapping to multiple cores can be supported with this separation approach too.

2.5.10.2 Separation on SW Level

Not all microcontrollers provide strictly separated register sets, respectively functionality of the hardware element (peripheral, core, memory). Typically those hardware elements require a common set of registers to control the functionality which cannot be atomically accessed .This is the case for several peripheral modules and peripheral features. Due to that a separation by software is required.

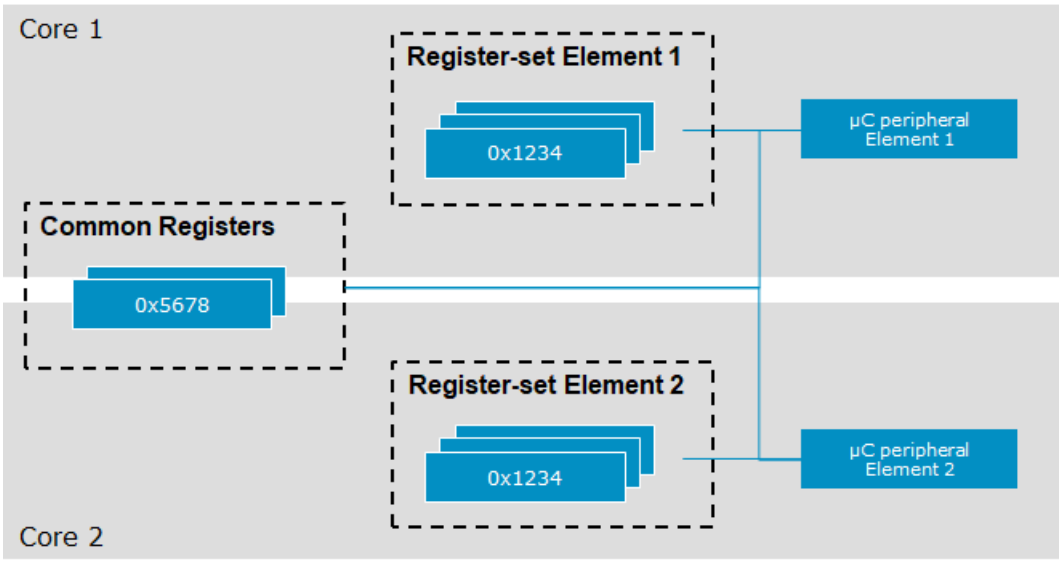


Figure 2.13: Shared Register-Set Separable on SW Level

For that purpose a software design pattern needs to be applied what can be in the worst case a spinlock (semaphore) in between the MCAL modules accessing the same hardware element from different cores. The performance impact of the exclusive area depends on the hardware element it shall be applied as well as the implementation of the spinlock. So for example hardware elements which are only written occasionally e.g. during start up or shutdown of the controller have a far less impact compared to "business" registers which are accessed frequently.

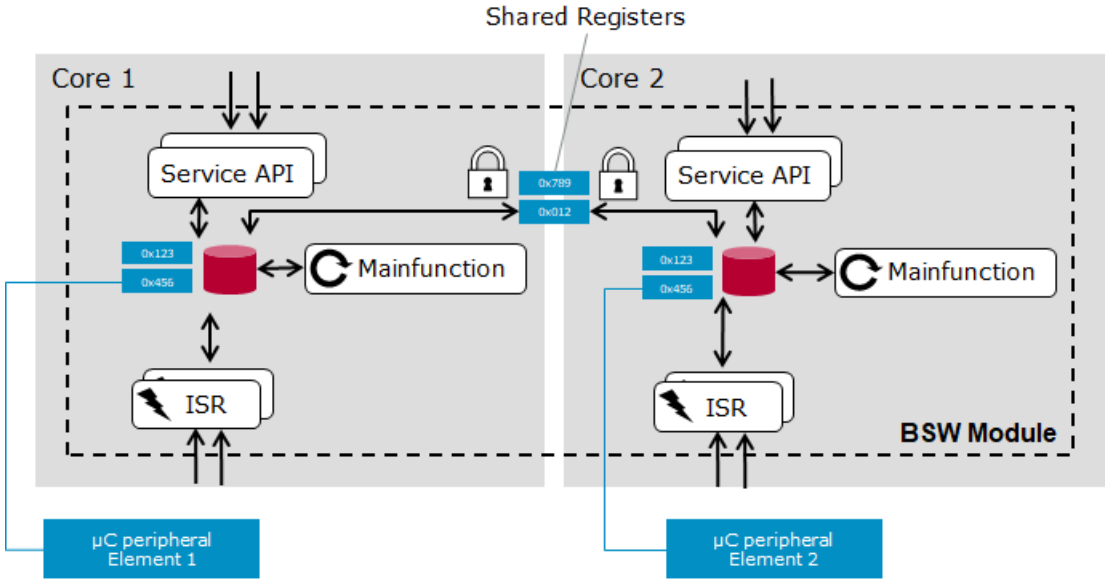


Figure 2.14: Separable Module on SW Level Example

An alternate solution to the protection of shared registers is to limit the access to one core only by finally changing the scope of the mappable element towards the next higher hardware element which allows an exclusive mapping to one core. Refer to Figure 2.13. For that purpose all accesses to the hardware element are coordinated by

one core while all cores transfer their requests using a messaging system (as IOC, but optimized to the MCAL needs). This use case requires that all service API implemented for MCAL modules dealing with such hardware elements behave asynchronously so that a no core is blocked by another one. As mentioned for the spinlock strategy above the implementation has a high influence on the performance in case it is done wrongly.

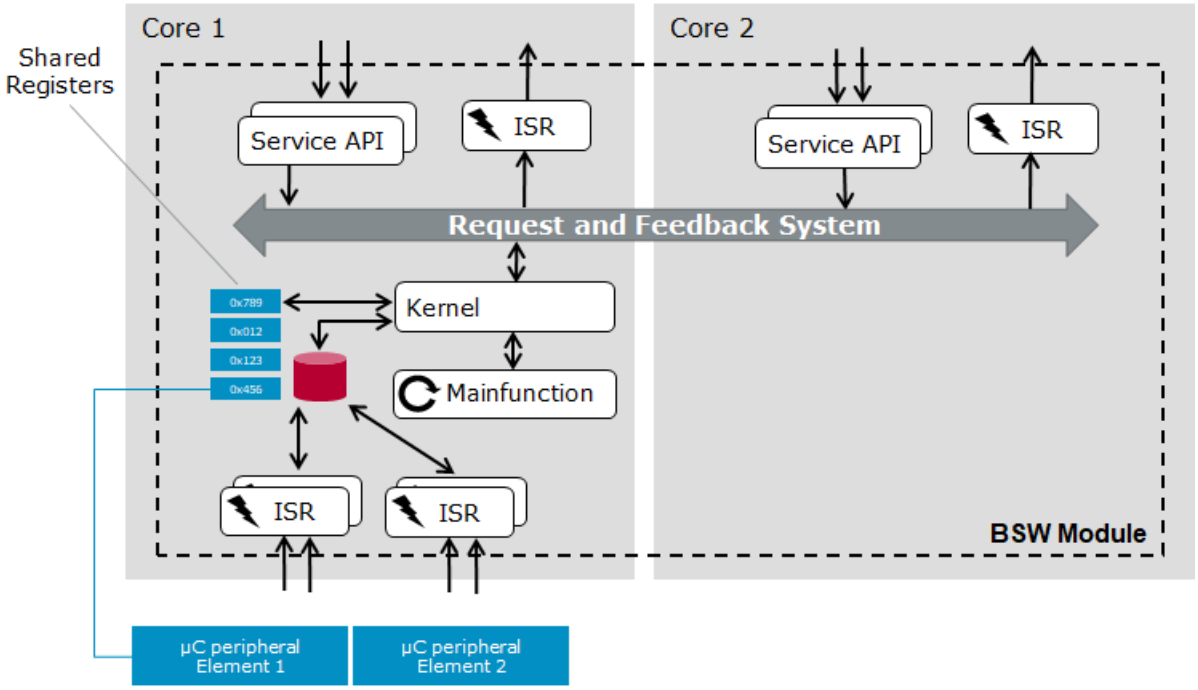


Figure 2.15: Separation on SW level Alternative Solution

2.5.11 Mapping of Elements

2.5.11.1 The Single-Core Module as Mappable Element

The mappable element is the MCAL module itself according to Multi-Core Type I. With this capability a MCAL driver does not provide any multi-core specific implementation and hence does not enable one of the new use cases. The reason for that is that the used hardware element does not allow any kind of concurrent access without a highly complex protection strategy.

Nevertheless the concept impacts the mapping of this MCAL driver as it is required to map the whole driver to a core. This is done by mapping its cyclic main function(s) and/ or interrupt routine(s) (if there are any of these) to exactly one OS Application.

By doing so, the driver is exclusively available on the core on which this OS Application is assigned to. As a consequence the scope of the MCAL driver becomes local. This capability is fulfilled by any standard single-core implementation.

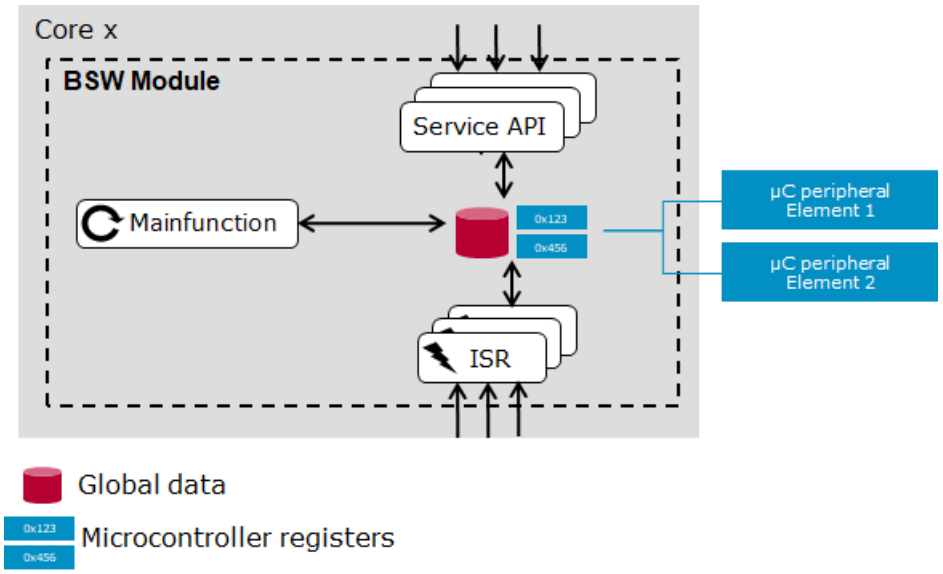


Figure 2.16: Mappable Element - Single Core Module

Figure 2.16 shows the simplified model of such a MCAL module. All core bound elements (service API, ISR and main function) have access to the according data (with local scope) and the microcontroller registers (mappable to this core). There is no separation regarding:

- Data (RAM, Register)
- Processing (Main functions)

All mappable μC elements (e.g. Timer channels) are handled by the same main function; all service APIs can control all μC elements. As a consequence the service API can be called by one core only.

The resulting mapping rule is: The module shall be mapped to one core only. Consequently the related hardware element is mapped to the same core only.

2.5.11.2 The Independent Hardware Element as Mappable Element

The mappable element is an independent hardware element such as a HW peripheral (e.g. CAN controller, Ethernet controller), core or memory which can be exclusively mapped to one core and consequently to one instance of a MCAL module. This mappable element is required to implement the described multi core type II, and also multi-core type IV, refer to MCAL Multi-Core Module Type II.

As a conclusion the related ISRs and service APIs are mapped to the same core too. If for example a peripheral has two independent peripheral modules, means elements (e.g. CAN networks) one is mapped to core 1 and the other to core 2. Each core only accesses the register set which is relevant for its peripheral element.

The same applies to the data of the MCAL driver which are now in local scope of the according driver instance. So data must be separated by element respectively core if it cannot be mapped 1:1 to the peripheral elements.

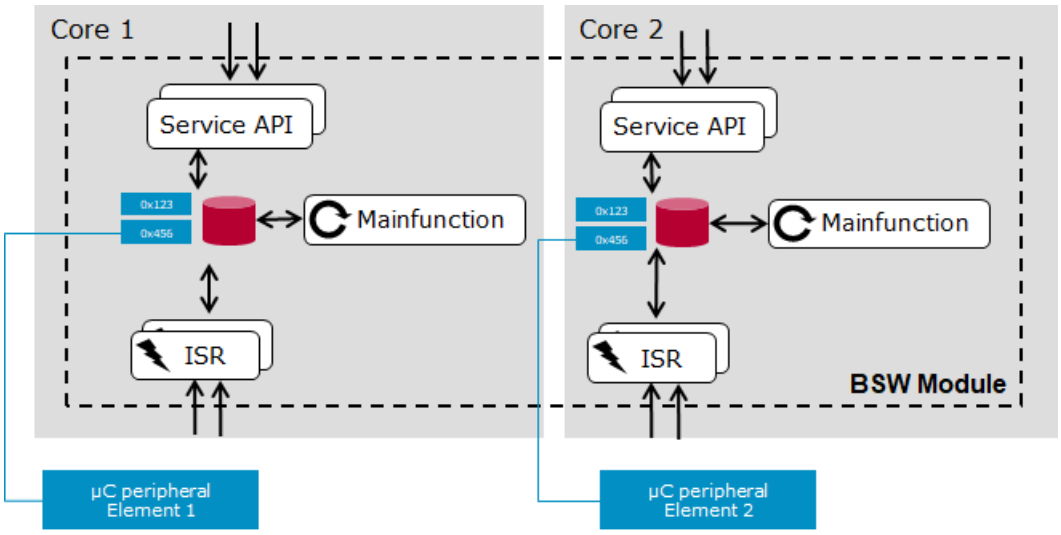


Figure 2.17: Mappable Element - Independent HW

This principle still applies if there are shared data and/or registers if they do not require exclusive access. The global driver status can for example be read atomically. Same applies to status registers which are readable without side effects.

From the behavioral point of view, MCAL modules realizing this principle appear to be instantiated multiple times, each instance includes a subset of the mappable elements but using a common code available with global scope.

The resulting mapping rule is: An independent hardware element shall be mapped to one core only. Consequently the MCAL module instance operating on the hardware element is mapped to the same core.

2.5.11.3 The Atomic Hardware Element as Mappable Element

The mappable element in this special case is an independent hardware element such as a HW peripheral feature which can be accessed atomically using the native access width of the hardware busses (e.g. 32bit for a 32bit microcontroller). This allows a mapping to several cores without the necessity to take care about concurrent access (e.g. DIO). Consequently this mappable element is required to implement multi core type III, described in section MCAL Multi-Core Module Type III.

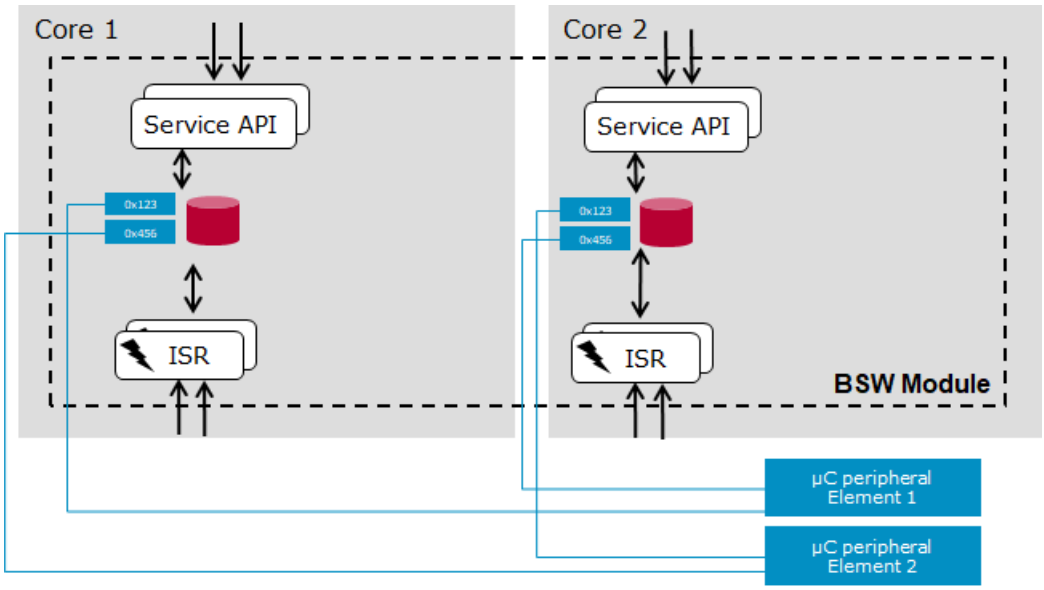


Figure 2.18: Mappable Element - Atomic HW

For this kind of mappable element there is typically a simple implementation available not implementing a main function as the access is done by the service API directly. In case data are used it is required that the access can be done atomic similarly to the mappable hardware element.

The resulting mapping rule is: An atomic hardware element can be mapped to any and even multiple cores. Consequently the MCAL module is mapped to the cores the hardware element is mapped to.

2.5.11.4 The Multi-Core-Module as Mappable Element

In this case the mappable element is again the MCAL module which can be mapped to at least one or multiple cores. The MCAL module itself is implemented according to multi-core type IV and applies one of the shown software separation strategies. However the service API are available on all cores the MCAL module is mapped to. The ISR of the MCAL module are ideally mapped to the core the user is running on. Typical MCAL modules are IO drivers required by any core implementing non atomic hardware elements, such as ADC, PWM, ICU, OCU and SPI.

The resulting mapping rule is: A multi-core MCAL module can be mapped to any and even multiple cores. Consequently all hardware elements are mapped to all cores the MCAL module is used on.

2.5.12 Examples

As a conclusion the MCAL distribution provides required service API to the cores these are needed. This is done depending on the multi-core type and the related mappable element. Consequently a few examples shall be shown below:

Example 1: DIO Concurrently Access by 2 IoHwAb

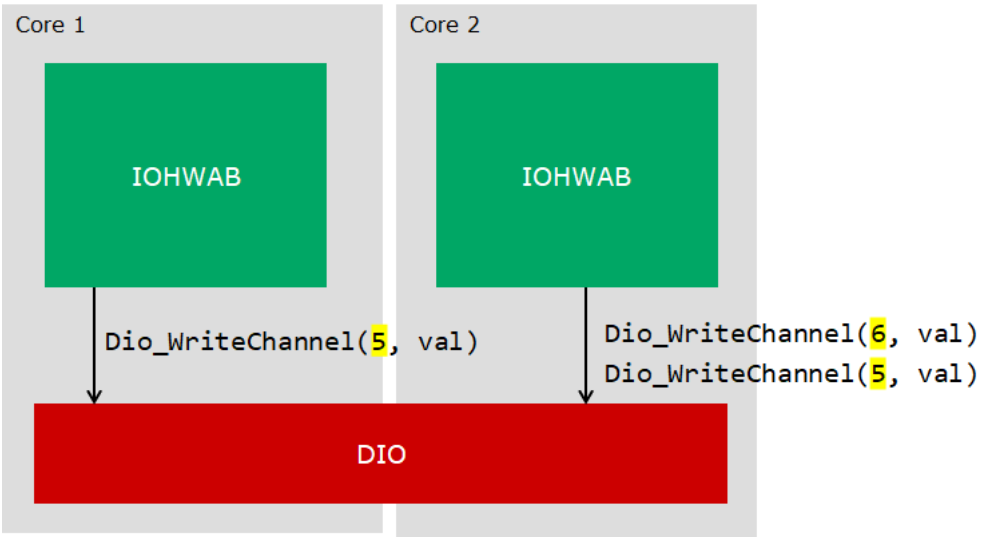


Figure 2.19: Example 1 - DIO - Concurrently Accessed by 2 IoHWAb on Different Cores

In the example Channel 5 of DIO is assigned to core 1 and core 2 whereas channel 6 is assigned to core 2. Each core contains an IOHWAB module. Both modules are allowed to directly call `Dio_WriteChannel` in their local core context. Limitation is that these only write to the channels assigned to the same core.

Example 2: DIO Accessed by 2 IoHwAb

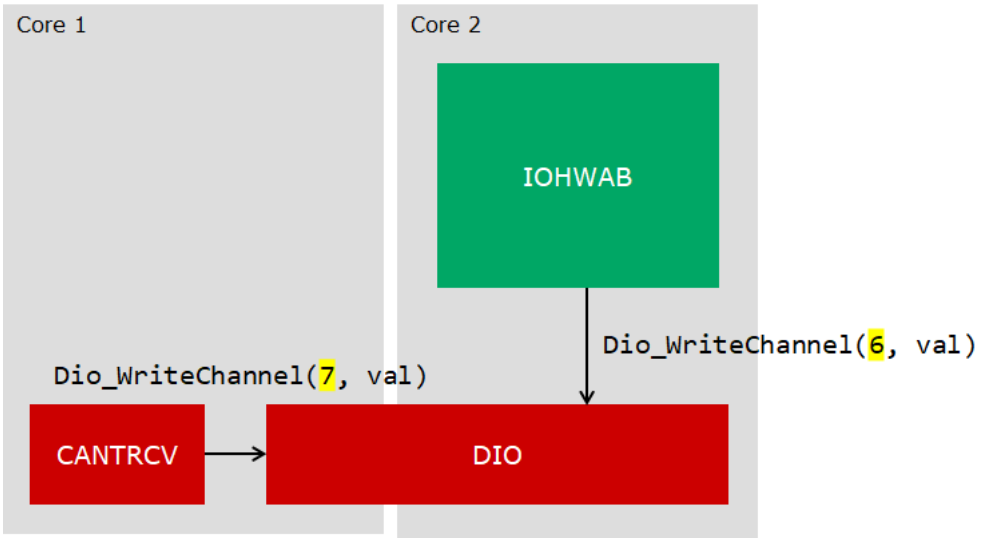


Figure 2.20: Example 2 - DIO - Accessed by CanTrcv and IoHWAb on Different Cores

As shown in the example, DIO is used by the CAN transceiver on core 1 and in the same time by the IOHWAB on core 2. Both DIO-users can directly call `Dio_WriteChannel` in their local core context.

Example 3: DIO Accessed By Master\Satellite Services

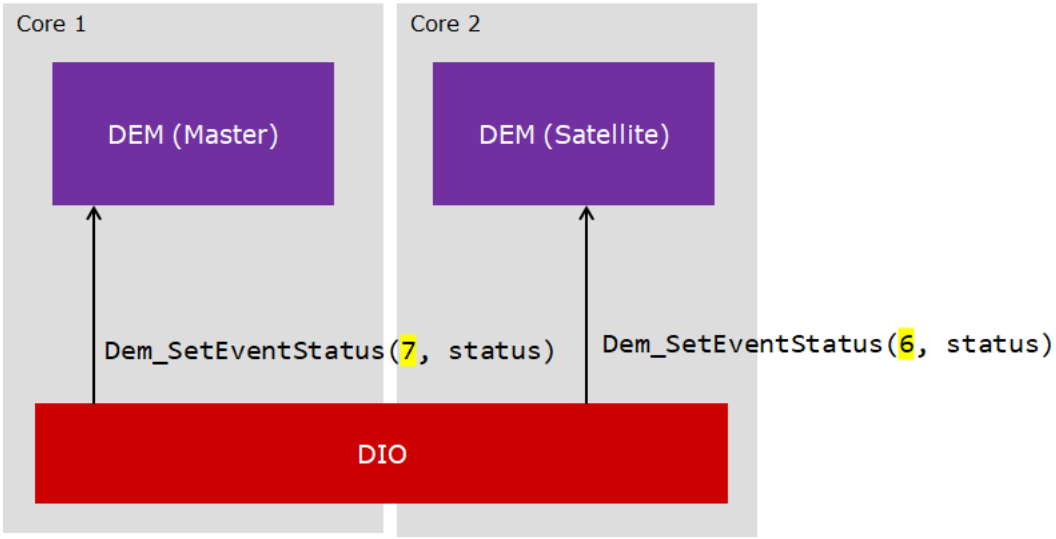


Figure 2.21: Example 3 - DIO - Accessed by Dem Master and Satellite

As shown in the example, DIO reports a diagnostic error to DEM. The call to DEM is issued on the core it occurs. DIO is not responsible to change the call context to another core. This of course requires that DEM provides its service API to the according core / partition.

2.6 Mapping Software to different Core Partitions

To be able to map symbols of a BSW module (or any other software) to the correct memory the memory mapping mechanism according to AUTOSAR_SWS_MemMap [1] shall be applied which offers an according multi-core support by introducing a core scope for the Software Addressing Methods. One can either apply GLOBAL (shared) or core LOCAL scope. Adding no scope, the mechanism assumes that GLOBAL scope is intended.

The following little example considers the core scope only not taking care about the additional safety partition information. Therefore, it is assumed that all examples consider QM software only. To comply with the latest standard 'QM' is added to the `SwAddrMethod` accordingly.

2.6.1 Allocation with Global scope

For GLOBAL (shared) allocation the implementer does not need to consider anything special as it is the default allocation making the allocated symbol visible within all cores.

Example:

Memory Allocation Keyword	Explanation
PWM_START_SEC_CODE_QM	Simple way to allocate global memory

According to the BSW module design it could be nevertheless interesting to map a certain amount of code differently, e.g. to map a library routine to a fast clone RAM area. One could do it in two different ways.

Option 1 - Different SwAddrMethod:

Memory Allocation Keyword	Explanation
PWM_START_SEC_CODE_LIB_QM	Library operation
PWM_START_SEC_CODE_USERIF_QM	User Interface
PWM_START_SEC_CODE_CTRL_QM	Control interfaces, e.g. Init, Shutdown

Option 2 - Different Prefix using the implementation extension 'IE':

Memory Allocation Keyword	Explanation
PWM_LIB_START_SEC_CODE_QM	Code of library implementation
PWM_USERIF_START_SEC_CODE_QM	Code of user interfaces
PWM_CTRL_START_SEC_CODE_QM	Code of control interfaces

Even if both options are technically feasible it is recommended to apply option 1 in the case the selected SwAddrMethods are re-used in an identical way across several BSW modules of one vendor, represented in a vendor specific SwAddrMethod catalog. Equipping the according SwAddrMethod with either standardized or vendor specific attributes one can easily apply an automated Memory Mapping later on. Therefore, a generic mapping can be applied.

Option 2 instead is not the recommended way of doing as the SwAddrMethod applied is the same and so identical options are provided so that the Memory Mapping will require user interaction to map according to the Prefix. Consequently, a specific mapping is required. However, there might be cases where it does not make sense to introduce very special (almost module specific) SwAddrMethods in a generic catalog. Instead, option 2 might be applied.

2.6.2 Allocation with Local scope

According to the distribution concept, it is required to map symbols to a special core, which is not known at software development time but instead at integration time. Means during software development the author needs to express the need of local allocation.

Example:

Memory Allocation Keyword	Explanation
PWM_FEATURE_A_START_SEC_CODE_QM_LOCAL	A special feature to be mapped to a specific core independently of feature B
PWM_FEATURE_B_START_SEC_CODE_QM_LOCAL	A special feature to be mapped to a specific core independently of feature A

According to the description given at the global allocation section it is recommended to keep a vendor generic SwAddrMethod 'CODE_QM_LOCAL' which is mapped according to the prefix. Means during the Memory Mapping process the core scope attribute 'coreLocal' can be considered accordingly.

When mapping the integrator needs to map FEATURE_A and FEATURE_B to the according cores.

So as a bottom line the use of the LOCAL suffix in the memory allocation keyword as well as the use of the core scope attribute 'coreLocal' in the SwAddrMethods express the intent of the software developer that a particular piece of code and/or data shall be mapped to core local memory. By using different Prefixes in the memory allocation keyword (e.g., FEATURE_A and FEATURE_B in the above example) the software developer gives the integrator the possibility to decide upon integration time to which core the piece of code and/or data effectively will be mapped to.

For generated configuration code, it would also be possible that a generator already knows the core the symbols shall be mapped to. Consequently, the following example would be feasible too:

Example:

Memory Allocation Keyword	Explanation
PWM_CNF_CORE0_START_SEC_CODE_QM_LOCAL	Generated configuration code to be mapped to core 0
PWM_CNF_CORE1_START_SEC_CODE_QM_LOCAL	Generated configuration code to be mapped to core 0

2.6.3 Allocation using Cloning capabilities

Special hardware architectures offer in addition the possibility of true memory area cloning, means those provide a local memory on each individual core at an identical global address. So, one could - for example for Multi-Core-Type II / III implementations - use a symbol with an identical name and address on each core. However, later-on the individual cores would act on the according individual local memory. Those symbols need to be allocated with a global SwAddrMethod or a specific prefix.

Option 1 - Different SwAddrMethod:

Memory Allocation Keyword	Explanation
MFL_START_SEC_CODE_CLONE_QM	Something to clone

However, when using a special SwAddrMethod for the purpose of CLONE one needs to consider that this one is hardware dependent and may not be supported on all platforms. So consequently, the integrator would need to remove the cloning capability by mapping it elsewhere. Especially for variables, this cannot be easily done as the symbol would automatically become GLOBAL and one would cause according resource access conflicts. Instead, a change of the code would be required.

As a conclusion, a different prefix shall be applied instead.

Option 2 - Different Prefix using the implementation extension 'IE':

Memory Allocation Keyword	Explanation
MFL_ADD_START_SEC_CODE_LIB_QM	Add operations - which might be cloned to speed up performance - if supported by HW
MFL_MUL_START_SEC_CODE_LIB_QM	Multiplication operations - which might be cloned to speed up performance - if supported by HW

Finally, the prefix usage allows to implement a generic software which can be either cloned or not independently of the used SwAddrMethod. Therefore, the integrator can finally decide what to clone or what to allocate in global memory. But note that also in this case the cloning of data needs to be considered inside the implementation what might lead to a variant of code (e.g. by conditional compilation). So the cloning can be recommended most likely for library implementations, or highly performance critical items such as interrupt nesting counters of the OS.

Note: In case a software designed for cloning is mapped to global memory the linker will cause an error for all variables originally intended for cloning. This is caused by the fact, that the variable would be instantiated several times with the same namespace.

2.6.4 How to determine the Core Scope?

The pattern on how to determine the core scope of symbols to be allocated is shown basing on the MCAL distribution concept. The idea can be applied to any other BSW module similarly.

2.6.4.1 Determining the Core Scope of MCAL symbols

Before expressing the core scope using the memory allocation mechanism one needs to identify what symbol of the module implementation (e.g. code, constants, variables) shall be available with what core scope. This can be done by associating an according symbol to a mappable element as described in the according BSW module specification following the MCAL distribution concept.

In the following step one needs to determine the number of partitions the mappable element can be mapped to. If the mapping targets a single partition only (1:1 mapping) then the symbols shall be considered as core LOCAL. If the mapping targets several partitions (1:n) then the scope shall be considered as core GLOBAL.

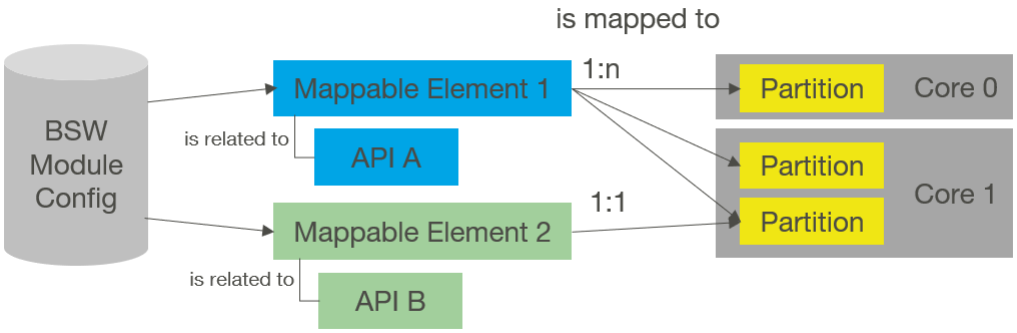


Figure 2.22: Mappable elements, related APIs and partition mapping

In Figure 2.22 one can see two mappable elements where element 1 can be mapped to multiple partitions and element 2 can be mapped to one partition only. Element 1 is associated to API A and element 2 to API B. Consequently, this would result in API A as GLOBAL and API B as LOCAL scope.

Applying the rule on the according MCAL implementations one needs to consider that one is not able to give a detailed recommendation for each MCAL API because the core scope might differ depending on the kernel implementation. So, it is at the end implementation specific.

2.6.4.2 Applying MemMap to the according Multi-Core-Types

The following table summarizes the typical resulting scope according to the MultiCore-Type definition for code, constants and variables. Other memory types are design specific or use typically a global memory scope, such as calibration data.

Multi-Core-Type	Feature / Type	Code / Constants	Variables
Type I (single core / legacy)	Legacy	GLOBAL	GLOBAL
	Single Core	LOCAL	LOCAL
Type II (running on core local data)	-	GLOBAL	LOCAL
Type III (usable from any core)	-	GLOBAL	GLOBAL
Type IV (core local kernel)	Kernel	LOCAL	GLOBAL
	User API's	GLOBAL	

Note: Older existing AUTOSAR BSW implementations usually comply with MultiCore-Type I or III (libraries). Especially for type I, the global memory allocation is used as this was state of the art before. Therefore, it is possible for backward compatibility to allocate Type I modules using the GLOBAL `SwAddrMethod`, but one shall consider that other cores might call the operations due to the global scope.

2.6.4.3 Allocating Driver internal Symbols

The allocation of MCAL driver internal symbols depends on the design and the way how these symbols are used by either public or internal interfaces. One can finally define the following rules:

1. The core scope of the allocated symbol shall have the scope GLOBAL if one of the using APIs has GLOBAL scope. Otherwise it shall be LOCAL. This rule applies only for CODE, CONST, VAR_INIT and VAR_CLEARED.
2. The partition scope might be limited further by appending a special memory allocation keyword, or an according safety integrity level information.

2.6.4.4 Allocation Example for Multi-Core-Type IV

Taking the example previously shown in this document, one can find a typical multi-core-type IV example with LOCAL as well as GLOBAL scope not showing the partition approach here.

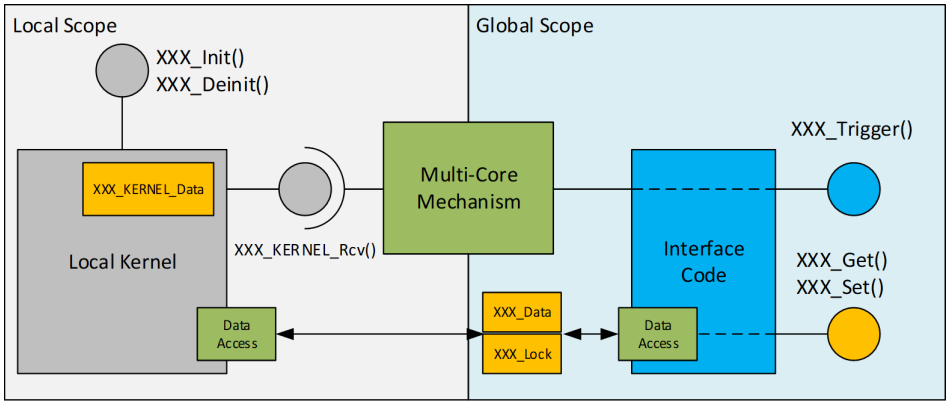


Figure 2.23: Multi-Core-Type IV Driver Example

The GLOBAL interface operations which are usable by any core and partition have similarly the GLOBAL scope. These are:

- `XXX_Trigger()` - `XXX_SEC_CODE_...`
- `XXX_Get()` - `XXX_SEC_CODE_...`
- `XXX_Set()` - `XXX_SEC_CODE_...`

The data elements used by these GLOBAL interfaces including the spinlock are global too as these need to be accessible by any core. These are:

- `XXX_Data` - `XXX_SEC_VAR_INIT_...`
- `XXX_Lock` - `XXX_SEC_VAR_INIT_...`

Instead the LOCAL interfaces, such as the control interface and internal interfaces as well as LOCAL data have the core scope LOCAL. So, these symbols can only used by the core these are mapped to.

- `XXX_Init()` - `XXX_CONTROL_SEC_CODE_..._LOCAL_...`
- `XXX_DeInit()` - `XXX_CONTROL_SEC_CODE_..._LOCAL_...`
- `XXX_KERNEL_Data` - `XXX_KERNEL_SEC_VAR_INIT_..._LOCAL_...`
- `XXX_KERNEL_Rcv()` - `XXX_CONTROL_SEC_CODE_..._LOCAL_...`

Please note that the shown example is design specific and might be implemented differently.

2.6.4.5 Allocation Example for Multi-Core-Type II

Another interesting use case is a driver of multi-core-type II which is working on core individual data. These data (and similar the HW peripherals) need to be accessed by the GLOBAL code when running on the according core. Therefor one needs to pass a pointer to the core specific data or even configuration structure when initializing the driver. This configuration can then be stored to a cloned pointer which exists on a per core memory but with a global address so that the driver code seems to access a global instance.

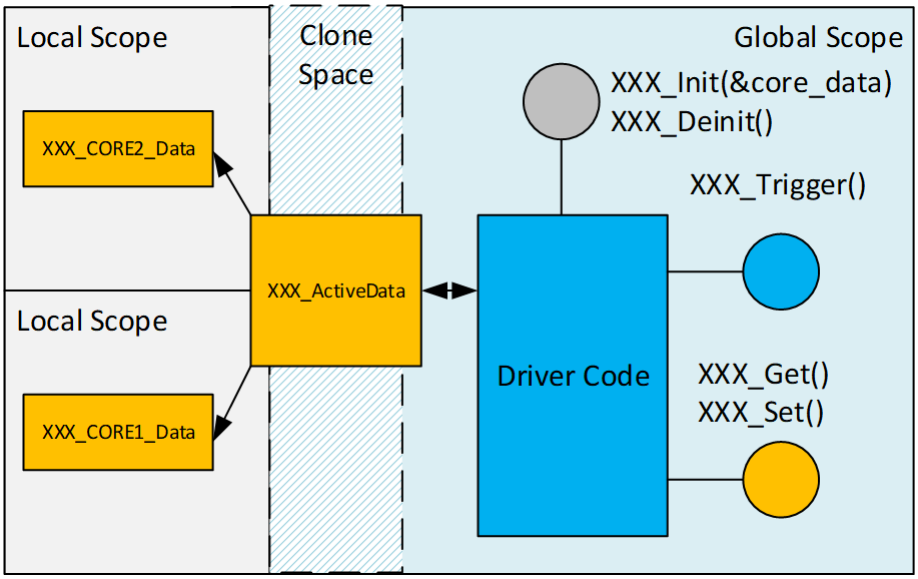


Figure 2.24: Multi-Core-Type II Driver Example

All code of the driver is generally allocated with GLOBAL scope:

- `XXX_Init()` - `XXX_SEC_CODE_...`
- `XXX_DeInit()` - `XXX_SEC_CODE_...`
- `XXX_Trigger()` - `XXX_SEC_CODE_...`

- `XXX_Get()` - `XXX_SEC_CODE_...`
- `XXX_Set()` - `XXX_SEC_CODE_...`

The Pointer to the active data instance is also allocated with GLOBAL scope:

- `XXX_ActiveData` - `XXX_SEC_VAR_INIT_CLONE_...`

Instead the core individual data (this can be also constants) are allocated with LOCAL scope and shall be accessed on the core these are allocated only.

- `XXX_CORE1_Data` - `XXX_CORE1_SEC_VAR_INIT_..._LOCAL_...`
- `XXX_CORE2_Data` - `XXX_CORE2_SEC_VAR_INIT_..._LOCAL_...`

Please note that the shown example is design and hardware specific and requires the support of the cloning capability.

2.7 Com-Stack Distribution

The proposed solution will provide concrete means to improve the usage of AUTOSAR BSW on multi-core microcontrollers.

Only minor impact on backward compatibility because this concept introduces mainly new optional functionality.

2.7.1 Introduction

The increasing use of multi-core processors makes it more and more important to efficiently use the provided cores of the controllers. Limiting the deployment of huge basic software packages like the Com-Stack to one single core is heavily limiting the effective use of the microcontroller's resources. Therefore, the distribution of the Com-Stack is a key for further basic software distribution.

The Com-Stack distribution requires specification changes in several BSW modules but shall not break existing contracts between modules. It is one goal of this concept to keep existing functionalities. The concept shall only extend the current functionalities by additional mapping scenarios.

Functionalities (sequences) which are not or only hardly adaptable to the new mapping scenarios will be kept as they are. To avoid unexpected behavior, limitations will be created which limit the mapping to the AUTOSAR scheme.

The main impact on interface level will be the change of some APIs from synchronous to asynchronous. For instance, when a job triggered on one core needs to be executed on another core and the initiating core should not be forced to wait (i.e. in a spinlock) until the job has finished.

The main benefit of this concept is the increased support for load distribution of the AUTOSAR stack by supporting additional distribution techniques.

2.7.2 Assumptions of Use

To apply the Com-Stack distribution several assumptions of use shall be given, to define the boundary conditions of the BSW environment:

1. The hardware implementation shall allow an individual mapping of peripherals to partitions or at least a dedicated core
2. It shall be possible to route hardware and software interrupts to one partition or at least a dedicated core (Hint: Routing of interrupts means to implement a kind of message passing, e.g. if the interrupt occurs on core1, but one needs to execute the code of ISR on core2 one needs to route the context from core 1 to 2 before the ISR is executed)
3. Single instance Crypto-Stack located in one AUTOSAR partition, where the whole Crypto processing takes place
4. The standard assignment of PDUs to Com instances does not consider ASW deployment but follows an assignment of PDUs according to the linked network type
5. Any interaction with EcuM needs to be done with the core-local EcuM instance

2.7.3 Constraints

To keep the complexity of the Com-Stack distribution in a manageable range several constraints need to be taken in account.

Constraints:

1. The Com-Stack distribution is limited to a setup, where each network specific stack can be allocated to one core
   (a) Channel individual distribution in order to distribute finer grained is not supported by AUTOSAR yet
   (b) CAN XL is an exception to this rule. Here a setup shall be supported, where a single CAN XL driver serves both the CAN stack and the Ethernet stack, which could be allocated to different cores.
2. A multi-partition (multi-application) AUTOSAR operating system is required to support the BSW distribution use cases
3. There shall be one partition, where the central modules for communication related state, mode and network management are assigned to (in this partition an EcuM instance shall be available as well).

4. There shall be one single SecOC instance which can be allocated to one core
5. TriggerTransmit (e.g. LIN, FlexRay) shall only be used "core local"
   (a) i.e. each TriggerTransmit PDU has to be assigned to the same partition, where the according network type is located at (no context switch between Com and <Net>If)
   (b) Gateway use cases are excluded from this constraint (cross core routings supported, as the PDU buffer is managed by PduR)
6. Com assumes signal related API calls always occur in the partition, where the affected signal (Pdu) is located in
   (a) Rationale: It might be possible for a software component to send data to another ECU via a PDU, which might be located in a different partition than the sending software component. In this case, Rte shall take care to switch the context to the target partition before calling Com APIs.
7. The entire J1939 stack (all J1939 modules) shall be assigned to the same partition as the DEM (due to the wide API between J1939DCM and DEM).
8. In a setup with a distributed Com-Stack, the post-build time configuration feature is not supported.
9. The APIs of the BusMirroring module shall only be called in the partition to which BusMirroring module is mapped to (via `MirrorEcucPartitionRef`).

2.7.4 Functional Elements

2.7.4.1 I-PDU configuration in a distributed environment

In an environment with a distributed Com-Stack, the network dependent module clusters are mapped to different cores (e.g. Can is mapped to Core 0, FlexRay to Core 1 and Ethernet to Core 2).

To manage this setup in the Com and IpduM module in an efficient way, individual main functions per network type can be defined, i.e. Com_MainFunctionTx_Can/Fr/Eth, Com_MainFunctionRx_Can/Fr/Eth, IpduM_MainFunctionTx_Can/Fr/Eth, IpduM_MainFunctionRx_Can/Fr/Eth.

All PDUs are assigned to the available main function instances according to the network type.

In this setup the PduR as the central instance for routing PDUs can call the target of each routing path directly, means RxIndications and TxConfirmations can be handled on the local instances as source and target are always located in the same partition.

For PDUs with direct transmission, it is recommended to assign them to a Com/IpduM instance located in the same partition as the according network. This is however, no limitation, as PDURouter would care to transfer the according I-PDU to the right

target partition, in case upper- and lower-layer connection is not assigned to the same partition. For sure, there are drawbacks if doing so, mainly affecting the system performance and resource consumption, but in principle, it is not limited by the approach.

In case TriggerTransmit handling is used, the according PDU must be assigned to a Com/IpduM instance located in the same partition as the according network (i.e. the assignment of TriggerTransmit PDUs to a Com/IpduM instance located in another partition, as the according network is not supported at all).

A special case is the Large Data Path of Com, as it is just a pure forwarding mechanism, which has neither buffers nor tasks and thus no own execution context.

Therefore, the assignment of PDUs to partitions (via MainFunctions) like for standard Com does not work here and therefore needs to be covered more individually.

One approach could be to deal with data, that is transported via a Large Data I-PDU, in the UL (most likely Rte) only in the partition, where the underlying network type is assigned to. In this setup, PduR would forward the I-PDU without dispatching.

Another approach would be to inform PduR in which UL context the Large Data I-PDUs will be handled (e.g., where `Rte_ComCbkRxIndication_<sn>` API shall be called; where `Com_Transmit` is called from UL (most likely Rte)). This information can be provided to the PduR via the Pdu to partition assignment (`EcucPduDefaultPartitionRef` or `EcucPduDedicatedPartition` for Com module) on basis of the EcuC Pdu.

In this case, the PduR will dispatch, if upper and lower layer connections are assigned to different partitions.

2.7.4.2 Pdu gateway

In general, one major job of the PduR is to take care of PDU routings between different networks. The principles how to master this in a multi-core environment are described in the following section.

In case the network specific module clusters are assigned to different cores, the Pdu R needs to take care to bring the context from source (RxIndication) partition (core) to target partition (core) and call the transmit API towards the lower layer only there.

In case *direct transmission* communication pattern is used, the PduR needs to cover following scenarios.

Routings just within the same network-type do not need any special treatment, means PduR can call the transmit API in context of RxIndication directly (no PDU buffering within PduR required).

The same handling can be applied for routings between different network types assigned to the same EcuC partition.

Routings between different network types assigned to different EcuC partitions (cores) require special treatment. In particular this means the PduR has to buffer the received PDU (in a shared memory area) and provide an execution context on the target core, where the PduR must call the transmit API for the buffered PDU.

Pdu gateway routings with *TriggerTransmit* handling need to be managed in a slightly different way as the PDU is provided to the lower layer Interface module in the context of the PduR TriggerTransmit callback.

Due to this pattern the PduR needs to buffer the received PDU in any case, even so the routing takes place within one network type.

Therefore, the required extension is limited to routings between different network types assigned to different EcuC partitions. In this case the PduR needs to switch the context from source to target, before calling the transmit API in addition to the normal routing mechanisms and take care to store the buffer in a shared memory area, which can be accessed from both contexts. The PduR needs to apply appropriate data protection mechanisms to guarantee data consistency even in a multi-core setup.

2.7.4.3 Connection to security stack via SecOC

In order to achieve an efficient treatment of the security relevant PDUs, the same principles like used for Com/IpduM with the split MainFunctions can also be applied to the SecOC and the crypto stack.

2.7.4.4 <Net>Tp Routing

In a gateway use-case for transport protocol modules, the `PduR_TpStartOfReception` is managed by PduR locally, means no inter-partition activity is triggered.

The behavior of `PduR_TpCopyRxData` and `PduR_TpRxIndication` shall depend on the routing variant.

In a direct gateway routing use-case, for transport protocol modules `PduR_TpCopyRxData` shall only copy data into Tp buffer (no calls across core boundaries shall be made).

PduR shall transfer the context to the target partition only within `PduR_TpRxIndication` call.

In a routing-on-the-fly gateway, use-case PduR shall transfer the context to the target partition either

- if `PduRTpThreshold` is reached within `PduR_TpCopyRxData` call
- if `PduR_TpRxIndication` is called before threshold is reached

In both flavors, TpTransmit API is called in context of target partition.

In case source and target are in same partition, the TpTransmit can be called directly inside the source partition.

2.7.4.5 Mode, state and network management

ComM and Nm shall take care for all interactions with <Net>Nm and <Net>SM modules, even so the network type is assigned to another partition. By doing so a centralized approach can be realized and the multi-core impact is limited to very few dedicated modules.

The kernels of ComM and Nm modules shall be assigned to the same partition in order to keep the interaction between these two central modules simple.

In addition, Dcm shall be assigned to the same core as ComM, so the mode APIs between Dcm and ComM can be kept as intra-partition communication.

In order to enable an efficient implementation of the NetworkManagement, the users of "Extra services provided by NM Interface" should call the respective APIs (e.g. `Nm_GetUserData`) only in the partition, where the underlying <Bus>Nm is assigned to. In this setup, synchronous inter-core calls can be avoided.

2.7.4.6 Startup/shutdown

EcuM calls towards other BSW modules shall only happen in case they are used in the pre OS phase. Otherwise, they shall be called via BswM, means by the partition-local instance of BswM.

2.7.5 Architectural Components

2.7.5.1 PduR as central inter-core dispatcher

The PduR shall cover all I-PDU related multi-core features caused by a setup with distributed network dependent module clusters in case <Net>If interfacing is used.

In order to ensure a high efficiency and performance the following types of PduR routing paths are excluded from the set of routing paths for which the PduR takes care of a potentially required core/partition transition. Thus, for those excluded routing paths the PduR does not need to check, if the source and the destination(s) of a routing path are assigned to different cores/partitions but can act like in a single partition setup.

- Transport protocol Interfacing
  - Transport protocol Reception/Transmission requires special means
  - Tp routing use cases are covered by Pdu Gateway
- Reception / Transmission of global time sync messages

  - Time sync messages must not be delayed, so standard PduR mechanism with asynchronous processing not sufficient
  - Special means for time synchronization need to be covered by StbM

As the PduR shall be capable to dispatch inter-partition communication, it needs to know the partition, to which each communication partner is assigned to. To provide this information, the Pdu configuration element in the EcuC module is augmented by the following configuration elements:

- `EcucPduDefaultPartitionRef` to provide a default reference to an `EcucPartition`.
- A list of `EcucPduDedicatedPartition` container to provide individual references to an `EcucPartition` on a per BSW module instance basis which override the default reference of the `EcucPduDefaultPartitionRef`. `EcucPduDedicatedPartition` in turn consists of the two elements `EcucPduDedicatedPartitionRef` referencing an `EcucPartition` and a foreign reference to the EcucModuleConfigurationValues of a BSW module named `EcucPduDedicatedPartitionBswModuleRef`.

Based on the above described partition assignment the PDURouter will assume an intra- or inter-partition routing.

2.7.5.2 Using EcucBusToPartitionMapping

The previous section explained how the PDU Router (PduR) can be configured for cross core routing using the `EcucPduDedicatedPartitionRef` (the partition reference). Routing between cores is requested by configuring different `EcucPduDedicatedPartitionRef`s for the source module (source partition reference) and destination (destination partition reference) of the PduR (see also [2] Chapter 7.11 *"Multicore Distribution"*). Note that this configuration does not require that the BSW Module configuration targeted by `EcucPduDedicatedPartitionBswModuleRef` exists one time per partition. The same module can be mapped to multiple partitions.

As there can be thousands of PDUs in one `EcuInstance`, there is a need to automate the creation of this configuration, which raises the question if these partition references can be deduced from other information sources.

ASW Side

If a PDU is used by ASW (via the Rte), this side's partition reference can be deduced by combining information from the ECUExtract with EcuC. This is done

by following the path from the Pdu via the ComUserSystemTemplateSystemSignalRef to a DataMapping, to a SystemSignal, further via the DataMapping to a Port and a SwComponentPrototype, and finally via the EcucPartitionSoftwareComponentInstanceRef to an EcucPartition (alternatively via SwcToApplicationPartitionMapping, ApplicationPartitionToEcuPartitionMapping and EcucEcuPartitionRef). Further optimizations might be possible by taking the scheduling into account (via a RunnableEntity's VariableAccess or ServerCallPoint).

Driver Side

The driver side's partition reference can be found via the EcucBusToPartitionMapping, which links a communication bus to an EcucPartition. It has three references:

- the EcucBusToPartitionMappingBusRef, which refers to the ComMChannel for that bus (on the ECU Configuration side),
- the EcucBusToPartitionMappingPartitionRef, a reference to the EcucPartition that the data path (e.g. bus driver, interface, etc.) is executed in, and
- the EcucBusToPartitionMappingPhysicalChannelRef, a reference to the PhysicalChannel for that bus (on the System Description side).

The ComMChannel was chosen to represent the bus, because it is part of the control path of a bus and is already used to represent the bus in other places (for example there is already an upstream mapping from ComMChannel to PhysicalChannel.commConnector). Please note that the control path does not have to be executed in the same partition as the data path, so it is possible that EcucBusToPartitionMappingBusRef refers to a different partition than the one that actually runs the ComMChannel.PhysicalChannel.

The reference EcucBusToPartitionMappingPhysicalChannelRef is optional. Via the PhysicalChannel's PduTriggerings, it makes it easier to find the PDUs in the System Description that are transmitted from or received within an EcucPartition.

If a PDU is received on one bus and routed to another bus (for example in a gateway), then both the source and destination side reference are an EcucBusToPartitionMapping.

2.7.5.3 Hints for partition assignment of I-PDUs

In case the PDURouter multi-core support is enabled, each module needs to take care to provide its partition assignment information for all referenced PDUs. The PDURouter

will check EcuC PDU configuration to find partition assignment for source / target of each routing path.

Some hints, how a default partition assignment can be defined:

- The default partition for a PDU is assigned based on its <Net> connections (<Net>If or <Net>Tp or <Net>Nm ...), as all network dependent modules are assigned to one dedicated partition.
- If default partition for a PDU is still undefined, do assignment according to Main Function reference
- If default partition for a PDU is still empty, the default partition needs to be handled individually during integration (not mandatory)

Rules for dedicated partition assignment

- Based on module features the dedicated partitions should be added to EcuC configuration module individually
- In case the module connection is on a different partition then the default partition assignment, it needs to introduce a dedicated partition container to EcuC Pdu config

For each routing path the PDURouter shall check, if

- A dedicated partition assignment for the connected module is found inside the according PDU configuration
- default partition assignment found

If partition assignment information of an I-PDU is missing, the PDURouter cannot judge properly which kind of routing path should be realized.

The PDURouter behavior for such cases shall be kept implementation specific, but at least a warning should be raised.

Recommendation would be to raise an error and inform the user about the missing information.

Examples for PDU partition assignments

- In case either source or target module of a routing path is a Large Data I-PDU the partition assignment depends on SWC deployment, as the Com has no own context for the Large Data Path and just forwards the call to/from PDURouter.
- For Com module take over partition assignment from Com internal MainFunction assignment to EcuC PDU configuration
- For IpduM take over partition assignment from Com internal MainFunction assignment to EcuC PDU configuration

Example: The COM module transmits an I-PDU via CanIf. CanIf is assigned to partition 1, means all `CanIf_Transmit` calls need to be executed in this partition. On

COM module level, the I-PDU is assigned to a Com_MainFunction instance running in partition 2. In this case, the PDURouter has to dispatch, as partition boundaries are crossed (inter-partition communication).

2.8 Crypto-Stack Distribution

In order to provide a load distribution amongst different partitions, the different parts of the Crypto-Stack shall be allocated to the different partitions. Hereby it shall be supported that such a partitioning happens on a crypto instance basis, i.e., the crypto driver instances shall be locatable onto different distinct partitions.

In order to support such a flexible allocation, the main threads of execution in the SecOC module (namely the respective MainFunctions) can be split into different Main Functions (at least one per partition).

Furthermore, also the Csm module can be split into different MainFunctions (at least one per partition).

In such a setup it is mandatory to assign all the Csm Queues to a dedicated Main Function to define the partition, where the respective Jobs has to be processed.

This way the flow through the crypto stack stays within the scope of a single partition and therefore does not require special multi-partition capable means.

The inter-partition communication between SecOC and PduR is managed by PduR.

In order to manage different timing requirements each MainFunction instance defines its time base individually.

This setup brings a couple of constraints for the configuration in order to run the crypto stack in an efficient way.

All `CsmJob`s, to which a `SecOC_MainFunction` points to, shall be assigned to `Csm_MainFunction`s which are assigned to the same partition like the respective `SecOC_MainFunction`. I.e. Csm processing of a Job needs to be handled in the same partition like SecOC processing.

In such a setup, Csm module can assume all `Csm_<Service>` API calls always occur in the partition, where the respective `CsmJob` is located in (thus no multicore extension for Csm's service APIs required).

2.8.1 Freshness value handling

In case the crypto stack is distributed across several partitions also the access to the freshness-manager SWC(s) is impacted, as multiple partitions need to access this SWC(s).

This is not beneficial for the performance (execution time) since there is a partition crossing involved.

There are three ways to deal with the freshness value in a distributed environment, which can be used in parallel:

- Leave as is and the RTE handles the partition crossing

  Note: Rather high impact on execution time

- Provide multiple freshness-SWCs (one per partition)

  Note: To achieve an efficient setup the entire processing chain (ComIpdu, SecOC PduProcessing, CsmJob, ...) should be assigned to the same partition

- Use the configurable C-API in SecOC (Configuration parameter: `SecOCQueryFreshnessValue`) and use a multi-core capable "library" instead

2.9 Multi-Partition Service Component Architecture Pattern

2.9.1 Introduction

Access to the standardized AUTOSAR interfaces of the BSW service components by the application SWC is supported by the RTE intra-partition as well as inter-partition. However, the implementation of a cross-partition operation invocation brings execution time overhead and adds latency between begin and return of the invocation in the application runnable. This may not be acceptable for applications that are highly time-critical and/or ECUs that have limited computational resources and cannot handle the overhead.

One alternative to mitigate these timing and performance drawbacks is to provide the required service interfaces in the same partition where they are required by the application SWC. Doing so allows the RTE to implement the operation invocation as direct function call with virtually no additional latency or execution time overhead. Communication between the partitions is now the task of the service components, which need to implement this service specific communication efficiently to gain any benefit from this architecture pattern.

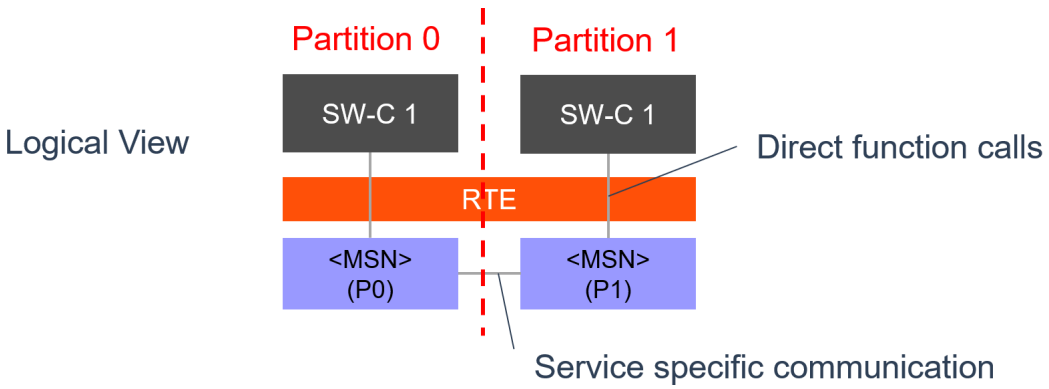


Figure 2.25: Logical view of multi-partition service modules

Following the general concept of the BSW multi-core distribution, the service component merely offers its services on multiple partitions. From a logical and execution context point of view, the service component exists (lives) in multiple partitions. That does however not mean, that the component's code or data must be multiplied. The same code can be executed on multiple partitions. Interfaces to neighboring BSW modules remain unchanged and are kept on one partition to maintain backward compatibility to the rest of the BSW stack. Potential existing requirements for BSW multi-core distribution that need to be fulfilled by the component will remain unchanged as well.

This architecture-pattern uses the AUTOSAR Master-Satellite-approach towards the application (RTE) while it keeps the API compatibility towards neighboring BSW modules.

If or if not a BSW service component implements this design pattern is implementation specific. Technically there is no need to do so because the RTE is capable to provide all required cross-partition/core communication between application SWC and BSW. Implementation of this pattern is hence merely an optimization.

2.9.2 Model Aspect

In the model-perspective, the software architecture looks like shown in Figure 2.26.

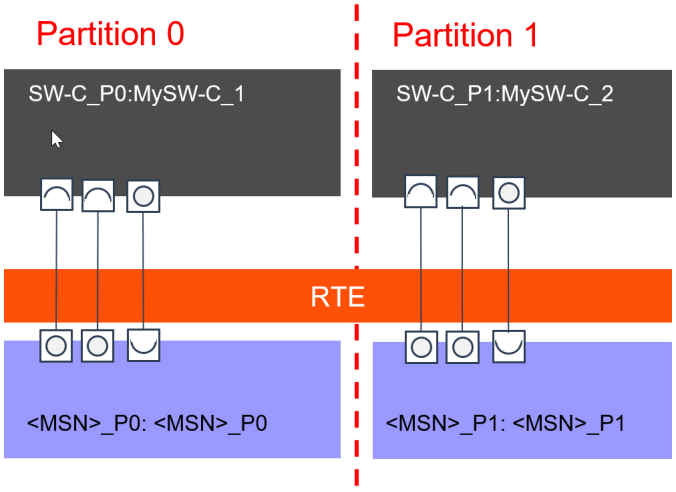


Figure 2.26: Model perspective on multi-partition service modules

The service BSW component provides individual Service SWC description types- and prototypes per ECU partition. Each of these component types provides the full set of service interfaces according to the current configuration of the component. It is recommended to add name-suffixes to the component type- and prototypes which show the partition assignment and thus help during integration.

Rationale for providing individual types is the AUTOSAR constraint that all prototypes of a component type must be mapped to the same ECUC partition.

The service interfaces are fully compatible. It shall be transparent to the SWC and the RTE on which partition they connect to a BSW service.

2.9.3 Implementation Aspect

For the BSW service component this pattern means that operations of the same service interface might be called simultaneously on multiple partitions on the same- or even different cores. One precondition therefore is that service APIs are specified to be reentrant. This attribute applies to single core- multi-threaded systems and allows concurrent invocation of the operations by application SWCs on different tasks. If it is not given (Reentrancy = Non Reentrant), it must not be overridden by this pattern by allowing concurrent invocation of service interfaces by application SWCs on different partitions.

Applying the pattern may also impact the runtime behavior of an API with respect to its "Sync/Async" attribute. A return of an synchronous API does not necessarily mean that the effect of the API has taken place but can also mean, that the API merely stored the function arguments and has triggered an internal processing request related to the API call. For example, the hypothetical API `<MSN>_RequestState(x)` is used to set the module's internal state to 'x'. If now the API is invoked on a different partition, it is legit that <MSN> buffers 'x' and performs the state change after the context has been switched back.

Callbacks of asynchronous APIs shall be invoked in the context of the same partition as the corresponding API. This means that a caller can remain unchanged, the BSW service needs to take care for performing any internal partition transitions.

How a component handles the concurrent access in multiple partitions is completely implementation specific.

A simple solution for the components is to use a `SchM_Call()`. However, since this is in the end the same mechanism as the RTE uses for core-transitions, such an implementation will not bring significant efficiency-benefits (latency, CPU overhead). The motivation for the BSW vendors is to come up with more efficient alternatives instead. Still, `SchM_Call()` might be an attractive option for sparsely used interfaces because it is easy to implement respectively use in the service component.

Another aspect are cyclic main functions of the service components. For some cases it may make sense to provide further, partition specific main functions, for others it will not. If additional main functions are provided, it is however recommended to add name suffixes that show the partition assignment and thus help during integration.

This pattern must cover also multi-partition use cases in which the BSW services are used by SWC in partitions with different ASIL. The component must ensure, that it handles the different partition contexts correctly, because the RTE will not do it. This also includes protection of the component's data. Measures must be taken that prevent corruption of data by a partition of lower ASIL (than any other that uses the service interfaces or C-APIs).

Figure 2.27 shows how the implementation can realize this pattern.

The service component provides common code that gets executed via the RTE on potentially any ECUC partition. On all partitions except Partition 0, the execution context is handed over to Partition 0. This is done implementation specific either synchronous or asynchronous.

Any interaction to other BSW components is done on Partition 0 exclusively.

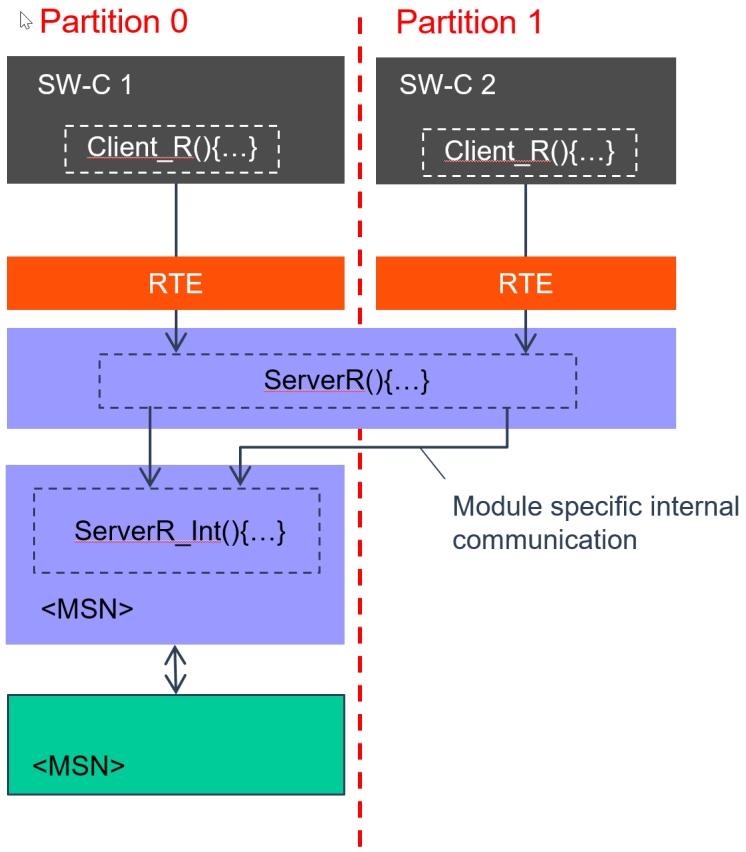


Figure 2.27: Implementation perspective on multi-partition service modules

For processing in the context of the different partitions, the component may provide dedicated cyclic main functions.

However, to avoid unnecessary effort in integration, the service components should restrain from creating elements (main functions, memory sections etc.) for partitions, on which its service interface is not needed.

3 BSW Distribution in Safety Systems

3.1 General overview on safety

In today's cars several ECUs may control safety relevant actuators depending on the functionality of the vehicle. Examples are electronic steering lock systems, adaptive cruise control systems or braking systems. If such a system shows a misbehavior a dangerous situation can occur where the driver is no longer able to drive the car in a save manner. To avoid such failures the specific ECUs must be developed in a way that the system can detect and react in a controlled way to such faults. The ISO 26262 [3] is the norm which describes how the development of such ECUs shall be performed to realize a save system. This norm defines four "Automotive Integrity Safety Levels" (ASIL) which classify the risk of the system. Based on the risks specific (safety) requirements of the system are derived. These requirements may be related to hardware (e.g. support for multiple channels to allow detection of hardware problems) or software (e.g. control flow checking) or both. In AUTOSAR we focus on software, so the hardware part will no longer be considered here. Be aware that an ASIL is always defined for a system, which means hard- and software, and with respect to software application software and basic software.

3.2 Safety solutions in AUTOSAR

AUTOSAR up to R4.1 supports safety systems by offering different base mechanisms which are typically required in such ECUs. The following list contains the main safety mechanisms:

- Partitioning of SWCs to support the isolation in space. This means that it is possible to separate SWCs of different ASIL from each other and to make sure that the SWCs are not able to write to other SWCs data. The realization requires hardware support (a memory protection or memory management unit) and is realized in the Os module and used by the Rte.

  Note: The general assumption in AUTOSAR is that preventing write access to other partitions is sufficient to achieve FFI (freedom from interference), while read access to other partitions is still possible. There may however be constraints or requirements that also demand a prevention of read access (e.g., see Chapter 3.2.10). In such a scenario certain solution approaches relying on read access are not possible (e.g., see Chapter 3.2.3).

- Timing and control flow supervision to monitor executing entities and to detect faults caused by blocking or wrong execution. In AUTOSAR the Os and the Wdg M take care of this issue.

- A safe communication via end-to-end protection is possible between ECUs (and even inside an ECU). This guarantees e.g. that the data which is send is not

modified between the sender and the receiver(s). The responsible module is the E2Elibrary.

Some other modules support additional mechanisms which are also useful in safety systems (e.g. CoreTest or RamTest).

The following picture shows how an AUTOSAR R4.1 can be used to support an ASIL ECU.

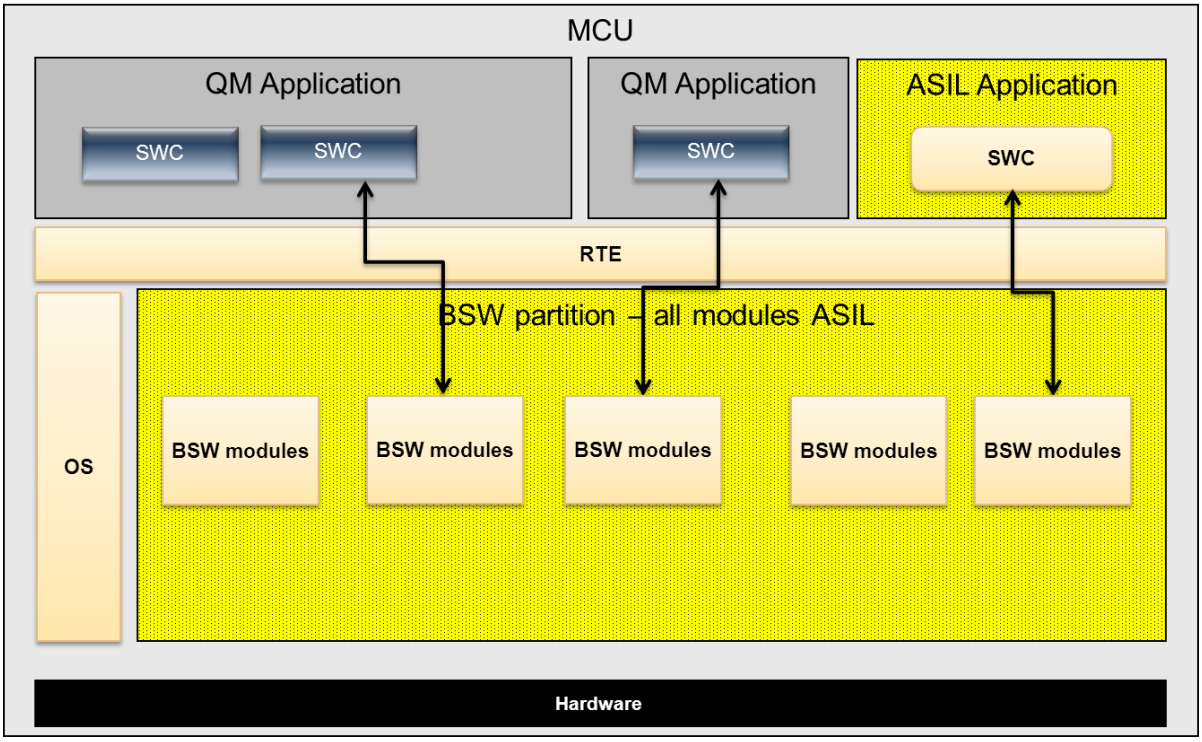


Figure 3.1: All BSW developed according ASIL

The approach works but has one big disadvantage: **all** BSW modules must be developed according the highest ASIL of the system. This causes a lot of additional work even if only some of the BSW modules are really required for a specific safety requirement.

Starting with R4.2 AUTOSAR offers an additional way how a safe system can be developed without the requirement to implement the whole BSW with the according ASIL. The key aspects of the new approach are:

- The BSW modules are not all mapped to one partition, but can be placed in separate partitions depending on the ASIL need. This means that a system can have several QM and ASIL partitions.

  Caution: Although it's possible to have more than one QM BSW partition per core, this should be avoided to be able to reuse existing QM BSW modules without modification. The reason is that these would need `CallTrustedFunction` wrappers for the cross-partition communication.

- The impact of the approach to single BSW modules is minimal. This means the scope of the modules is the same on ASIL and QM. There is no change of interfaces between modules.
- Only the modules which provide the safety relevant features (e.g. the memory protection offered by the Os) need to be developed according to the system's ASIL. Sometimes it is even possible to limit the required ASIL functionality to a subset of a BSW module.

The ASIL modules inside the ASIL partition(s) need to be specifically developed. They not only need to meet the requirements of the ASIL level, but they also need to detect if they are called from inside the partition or outside the partition.

With this approach it is possible:

- To reuse existing BSW modules which were developed on QM level (no ASIL) without module modification.

The proposed approach has to be assessed case by case in order to estimate the applicability of this approach for the particular safety case and the benefits of combining QM/ASIL modules compared to a pure ASIL approach.

BSW modules can be placed in different partitions. AUTOSAR supports several QM and ASIL partitions. The following figure shows an example mapping. Here the ASIL SWC has save access to some hardware via an own partition in the BSW which contains an IoHwAbs and the needed drivers below.

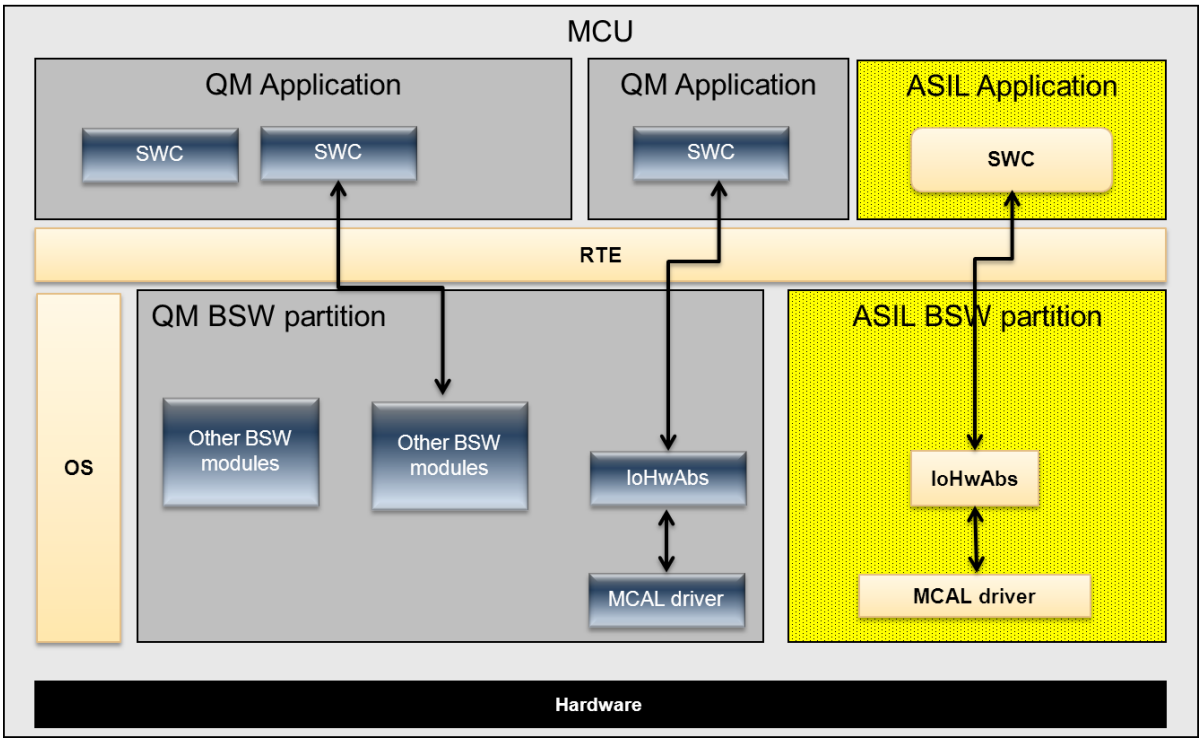


Figure 3.2: BSW modules mapped in different partitions

It is strongly recommended that QM BSW partitions run in user mode if possible in case we have BSW ASIL partitions in the system to avoid changes to hardware registers (e.g. MPU settings). If this is not possible (e.g. hardware supports supervisor mode only) you need additional means to assure freedom from interference.

3.2.1 Some modules are always ASIL

Since the protection mechanism is provided by some specific BSW modules (e.g. the Operating System) these modules have to be developed according to the highest ASIL in the system. If they are not developed at this level it cannot be assured that they are able to fulfill their supervision task. The decision which modules have to be developed to ASIL is always project specific and is determined from the safety requirements of the system.

3.2.2 Overall configuration

The separation of BSW modules in different BSW partitions for safety needs to be configured in the ECU configuration. The mapping is done in the EcuC and Os configurations.

For each such BSW partition an OsApplication is required. The following settings apply to the Os configuration of each BSW OsApplication:

Name	Value for BSW partitions
`OsTrusted`	`TRUE`
`OsTrustedApplicationWithProtection`	`TRUE` or `FALSE`
`OsTrustedApplicationDelayTimingViolationCall`	`TRUE`

Other attributes of the OsApplication can be filled as needed. Note that hook functions of BSW partitions have no meaning in AUTOSAR and shall be avoided.

Additionally note that the OSApplication TRUSTED attribute (`OsTrusted`) of the OSApplication is not related to ASIL/non-ASIL.

Afterwards the BSW modules, which are used, have to be configured and mapped to the different partitions. The mapping is done in EcuC:

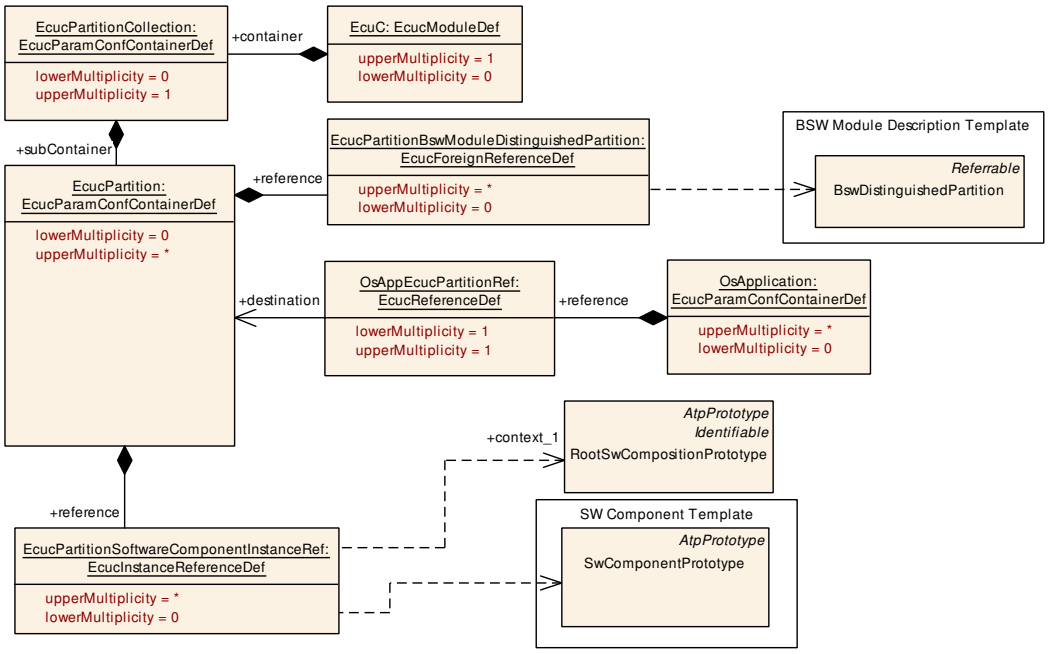


Figure 3.3: EcuC configuration - mapping of BSW to partitions

The `EcucPartitionCollection` (multiplicity 0..1) contains all partitions of the system. For each of them a sub container `EcucPartition` (0..*) exists which contains references (`EcucPartitionBswModuleDistinguishedPartition` (0..*)) to the BSW modules (via BSWDT) which are placed into this partition.

The `EcucPartition` configuration of each BSW partition contains a link to the OsApplication of this partition (`OsAppEcucPartitionRef`).

3.2.3 Crossing partition boundaries

When BSW modules are placed into different partitions, the crossing of boundaries is the biggest issue which needs to be solved. The following figure shows the scenario in a quite general view:

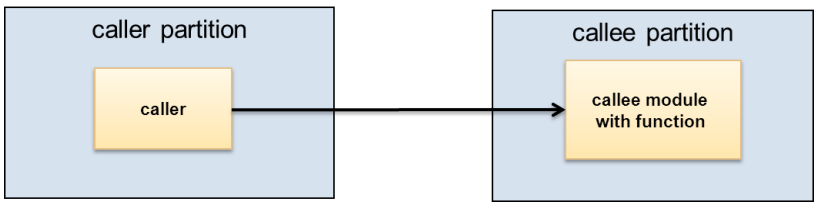


Figure 3.4: Cross partition call

This is due to the fact that the called service assumes that it has full access to module local data, which is not true if the call is performed from another partition because the memory protection settings are still those of the caller. In general there are 3 possibilities how the problem can be solved:

1. Instead of a direct call the caller can do an `ActivateTask` to a Task from the callee partition. In this case the activated Task will perform the real call to the function. Instead of the `ActivateTask` a `SetEvent` can be used as an alternative. Note that both mechanisms work in an asynchronous way which means that the original caller may need to wait or have to poll for the result
2. The caller can use `CallTrustedFunction` to enter the callee partition, or the callee after being called use `CallTrustedFunction` to hand over to its partition. After entering the function can be called directly. `CallTrustedFunction` makes sure that the caller gets the appropriate rights to make the call, e.g. changing the memory protection to the setting of the called function.
3. The call of the function may be directly possible if the called function does not write to own data or calls other functions which write to such data. E.g. if the function just reads out a value and return it. Basically, such a function behaves like a library.

Dependent on the mapping of the BSW modules to different partitions the right option has to be chosen. For all function calls between BSW modules located in different partitions which are synchronous, we will focus on the calling possibilities (2) and (3). Because as already stated QM modules are not changed, we have to encapsulate calls which are made from QM partitions to ASIL and vice versa. The ASIL module is always responsible to handle the boundary crossing since the QM module is not touched and does not know this border. This means that if the ASIL module is the caller, the boundary handling needs to take place on the caller side, and if the ASIL module is the callee, the boundary handling needs to take place on the callees side

The following descriptions focus on ASIL and QM BSW modules. Besides BSW modules also CDD might be included in the system. For CDDs the same rules and restrictions apply (if not otherwise explicitly stated)

3.2.3.1 QM modules calls ASIL

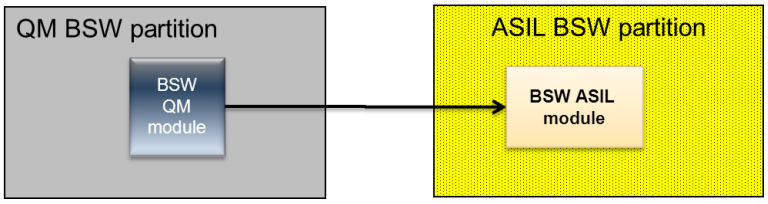


Figure 3.5: QM calls ASIL

As already stated the QM module which performs the call is unchanged. Even more: The QM not even knows that the called function (module) belongs to a different partition. This means we have to encapsulate the called function into a stub which performs the boundary crossing.

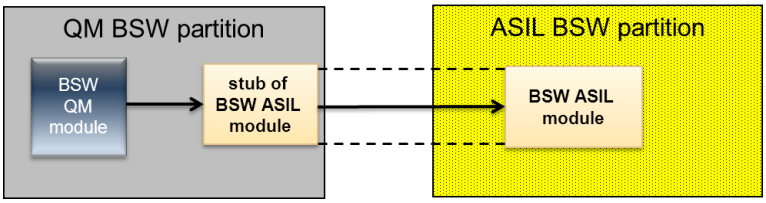


Figure 3.6: Details of QM calls ASIL

This stub function can be static or generated and belongs to the called module. It can be seen as a new function entry of the called function of the ASIL module. The following message sequence chart shows the calling sequence. As you can see the stub itself also has two parts, one on the caller side and one on the callee partition.

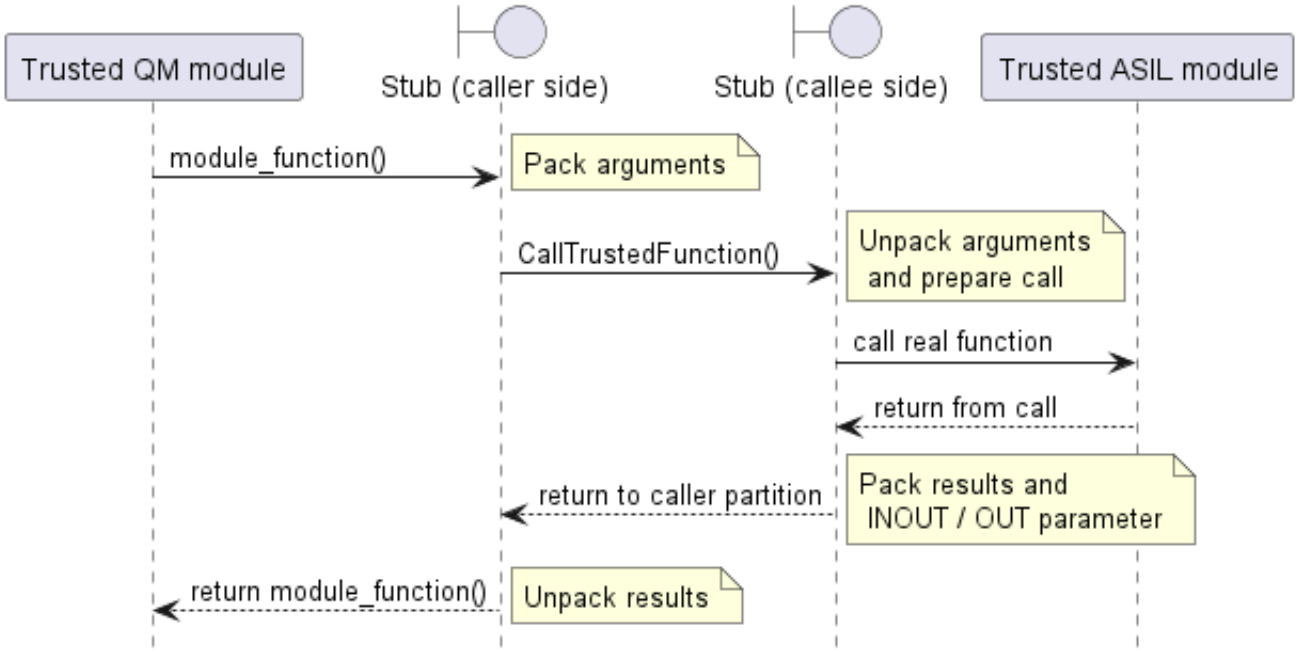


Figure 3.7: Call sequence when a stub is used

The stub itself can be static (hand written) or generated based on the available configuration information. The next two sub chapters are detailing the different approaches.

3.2.3.1.1 Static stub

A static stub has to cover all situations. In our case the important issue is to find the caller partition in order to make type of call. The next code fragment shows an example of a static stub:

```
1  StdReturnType module_function()
2  {
3      runId = GetCurrentApplicationId();
4      if (runId == module_applicationId)
5      {  /* direct call possible */
6          return Modulemodule_function_real()
```

```
7       } else {
8          CallTrustedFunction(MODULE_REALFUNCTION_ID,NULL)
9          ...
10      }
11  }
```

Note that you have to init your own module application Id (or use directly the generated application name)

3.2.3.1.2 Generated stub

If an optimized version of the stub shall be generated the generator needs all information (e.g. who calls the function) in order to create the best code. If information is missing or incomplete the generated code might either not be able to generate the code at all or the code may fail during runtime.

AUTOSAR has an abstraction for calls between different partitions. This method is used in multicore systems to allow modules a communication between different partitions on different cores.

The mechanism used by the generated code is offered by the SchM: `SchM_Call`. The `SchM_Call` will then be mapped within the SchM to one of the methods listed in Chapter 3.2.2.

For finding the best method for crossing the boundary the central question is:

Who will call the function (and use the stub)?

This information must be provided by the user via the SchM configuration. The configuration consists of caller, callee and references to their modules (and also implicit to the partitions). The following diagram from the RTE shows the configuration of `SchM_Call`:

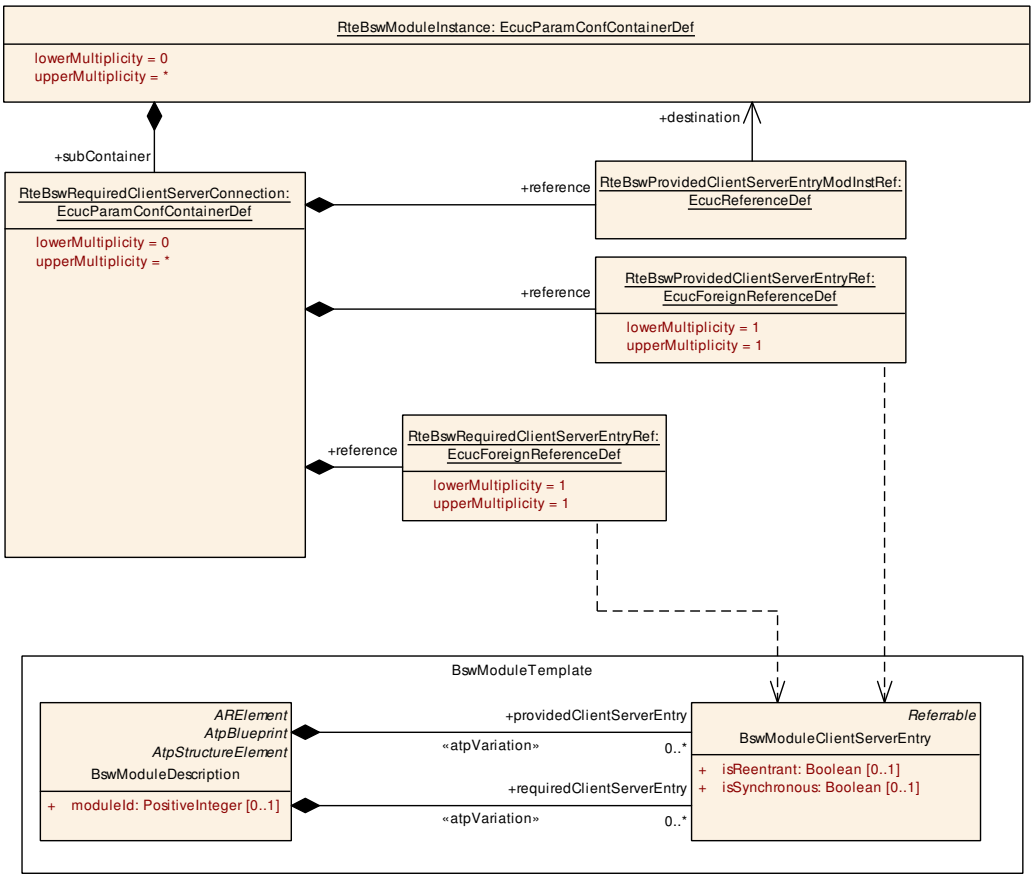


Figure 3.8: Configuration of SchM_Call()

Based on this information and the information where a BSW module is placed, the Sch M can generate optimized version of the `SchM_Call`.

E.g. if there is only one stub user and this user is placed in the same partition as the called BSW module a direct call is possible. Example of a stub using `SchM_Call`:

```
1  Std_ReturnType module_function()
2  {
3      Std_ReturnType r;
4      (void) SchM_Call_target_module_function(&r);
5      Return r;
6  }
```

The approach to generate a stub has some limits which need to be considered during system development:

- Calls from integrator code: A configuration via `SchM_Call` is not possible for integrator code since this code does not belong to any BSW module and does not have any configuration (EcuConfiguration) and module (BSWDT) information which could be used. In such cases a hand written static stub has to be used.
- A `SchM_Call` configures exactly one caller-callee relationship. If a function is called by different callers, the generated part of the stub cannot distinguish which `SchM_Call` is required for which caller. In such cases a static stub is required.

Note: If also the QM caller would use a `SchM_Call` instead of the real function name the stub could be avoided completely. But this would contradict the target to reuse existing QM code unmodified.

For parameter handling see Chapter 3.2.3.5.

3.2.3.2 ASIL calls QM partition

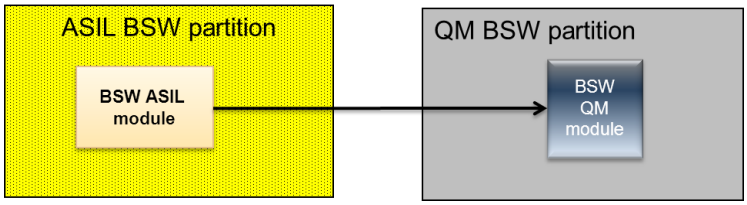


Figure 3.9: ASIL calls QM

This chapter covers now the direction of an ASIL caller and a QM callee. Here the ASIL module already knows that a boundary crossing is required. (Otherwise the called QM function would be an ASIL function). Since the QM function shall not detect any difference when called from an ASIL function or from a QM function in the same partition, it must be called as would the call be locally performed.

As a consequence of this we need again a code fragment which performs the real call. This code fragment in this case is named wrapper.

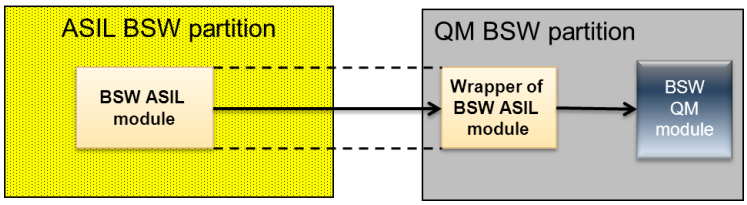


Figure 3.10: Wrapper for ASIL calls to QM

This wrapper function can be statically or dynamically generated and belongs to the caller module but is partly executed in the partition of the callee. The following message sequence chart shows the calling sequence, when `CallTrustedFunction` is used:

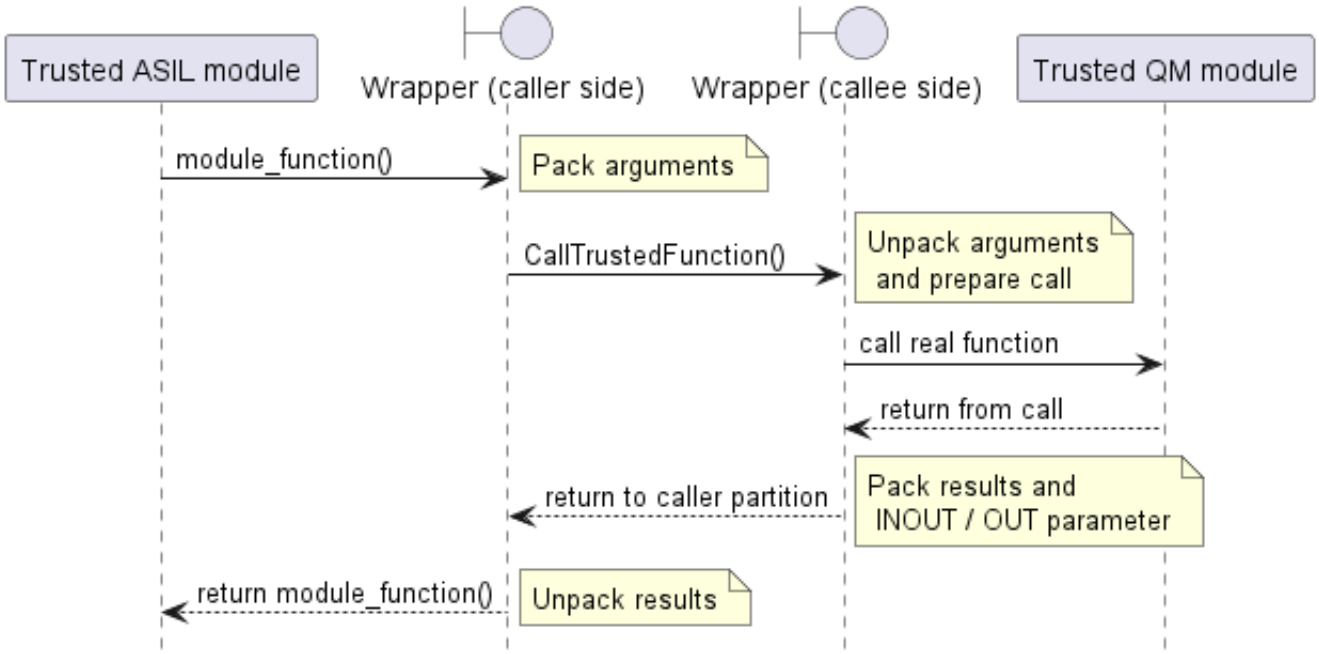


Figure 3.11: Call sequence when a wrapper is used

We can again differentiate in a static wrapper and wrappers which are generated out of the configuration.

Note that independent of the technical solution it needs to be checked whether such calls are allowed within the project specific safety goals.

3.2.3.2.1 Static wrapper

The following code fragment shows a possible wrapper in case only one "user" calls the function (in other cases the buffer handling needs to be extended).

In the example the `CallTrustedFunction` mechanism is used:

```
1 /* definition of parameter structure for CallTrustedFunction */
2 struct parameter_struct {uint8 return_value};
3 uint8 wrapper_function(void)
4 {
5    struct parameter_struct local_struct;
6    CallTrustedFunction(MODULE_REALFUNCTION_ID, &local_struct);
7    return local_struct.return_value;
8 }
```

This is the second part of the wrapper which is located in the callee partition:

```
1 void TRUSTED_module_function(TrustedFunctionIndexType a,
     parameter_struct *local_struct)
2 {
3    local_struct->return_value = module_function();
4 }
```

3.2.3.2.2 Generated wrapper

If the wrapper shall be generated the generator needs specific information in order to create the best code. If information is missing or incomplete the generated wrapper code might fail.

Like the stub handling In Chapter 3.2.3.1 we can use the `SchM_Call` service to hide the partition transitioning. In contrast to the stubs we need not to focus on possible users of the wrapper - the users are just the ASIL module functions - but on the called function. This means we have to find out the callees partition in order to make the right call. Similar to stubs, this information must be provided by the user via the SchM configuration.

There is also one limitation of this approach:

- Calls to integrator code: A configuration via `SchM_Call` is not possible since the integrator code does not belong to any BSW module and does not have any configuration (EcuConfiguration) and module (BSWDT) information which could be used. In such cases a separate static wrapper has to be used to encapsulate calls from integrator code and the integrator code need small changes, e.g. changing the name of the called function to avoid name clashes.

For parameter handling see Chapter 3.2.3.5.

3.2.3.3 ASIL calls ASIL

The case of an ASIL to ASIL call can be seen as a combination of Chapter 3.2.3.2 and Chapter 3.2.3.1. Also here a generic glue code might be needed if the modules are not placed in the same ASIL partition. In this case either the caller or the callee have to provide this glue code. In an ASIL system the glue code is normally provided by those modules which have the higher ASIL. The glue code can be created statically or can be generated.

For the generation of the glue code the following limitations exist:

- Calls to integrator code: A configuration via `SchM_Call` is not possible since the integrator code does not belong to any BSW module and does not have any configuration (EcuConfiguration) and module (BSWDT) information which could be used. In such cases
  - Either a static glue code has to be used to encapsulate calls from/to integrator code and the integrator code might need small changes, e.g. changing the name of the called function to avoid name clashes.
  - or offer vendor specific configuration parameter which holds per callout a reference to the OsApplication where the integration code is placed.
- If we know only the address of the callee (this can happen if the interface is generic and function pointers are used for the call, e.g. in the PDU Router) we

need a dedicated vendor specific configuration parameter for the ASIL module which provides the information in which partition the callee is located.

3.2.3.4 QM calls QM

This caller-callee combination is only supported if the involved BSW QM partitions share the same memory access rights or the BSW QM modules are modified to encapsulate cross-partition calls in `CallTrustedFunction` wrappers. This however violates the initial assumption for this approach to reuse existing BSW modules without modification.

3.2.3.5 Parameter passing

In the previous sections we showed how a call to a function in another partition can be made. Besides the real call mechanism there is another important topic and this is the passing of parameters to the callee and passing results back to the caller. The question behind this is: How does the callee access these parameters and how can results be propagated back to the caller.

AUTOSAR differentiates between IN, OUT and INOUT parameters which are passed. IN parameters are not critical, because they are normally passed by value and even for cases where a by reference passing is done the callee is not allowed to write to them. This means that they do not pass any information back to the caller.

OUT and INOUT parameters are used to return results from the callee back to the caller. The question now is: how can these values passed back to the caller if callee and caller are not in the same partition.

In general the following methods are possible:

1. If caller and callee are in different partitions the callee works on a copy (for INOUT data) or empty space (OUT data) and when returning back to the caller the values are copied back. For the inter partition communication of data AUTOSAR offers the IOC mechanism of the Os. However, often usage of IOC can be avoided by copying such that only read access is needed.
2. A hardware specific solution: In such cases a copy / extra buffer is avoided by using dedicated hardware features of the used microcontroller which guarantee freedom from interference. E.g. If the hardware allows for private shared memory areas between caller and callee.

In the following we will show how (1) works. Option (2) depends on the used hardware and is not standardized in AUTOSAR. The following code fragment shows an example how the parameter passing works (case: ASIL calls QM)

```
1 /* caller side code */
2 Std_ReturnType _Dem_GetOperationCycleState(
3     uint8 id,
```

```
    Dem_OperationCycleStateType* state)
{
    ...
    /* setup params struct with arguments */

    ret = CallTrustedFunction(GETCYCLESTATE,&params)
    if (ret == E_OK)
    {
       IocReceive_RETURNVALUEGETCYCLESTATE(&ret);
       IocReceive_VALUEGETCYCLESTATE(state);
    }
    return ret;
}
```

```
/* callee side code */
void TRUSTED_GETCYCLESTATE(TrustedFunctionIndexType a,
                           parameter_struct *local_struct)
{
    Std_ReturnType localreturn;
    uint 8         localid;
    Dem_OperationCycleStateType localstate

    /* setup parameters from local_struct */
    ...

    localreturn = Dem_GetOperationCycleState(localid,
                                             &localstate);
    IocSend_RETURNVALUEGETCYCLESTATE(localreturn);
    IocSend_VALUEGETCYCLESTATE(localstate);

    return;
}
```

Note that the above example is quite typical for AUTOSAR inter-partition calls. It assumes that the lifetime of the buffer is equal to the duration of the called function. If this is different, e.g. one function which just provides a buffer and another function at a later time indicate that the buffer is now ready (example: NvM read mechanism) an adoption is needed.

3.2.4 Access to peripherals / hardware

In AUTOSAR the access to peripherals or hardware is limited to BSW modules. Typically only some of them require a real access, e.g.:

- The Os switches between different contexts and need to read/write the context registers. Also interrupt locking requires normally access to hardware registers or execution of privileged instructions.
- During startup the Mcu driver needs to enable the microcontroller clocks and may perform further initialization of registers
- IO drivers need to access their part of the hardware.

- ...

If parts of the BSW are now running in a partition where the memory protection is enabled the full access to hardware is normally no longer possible. In such cases a hardware access can be realized by:

1. "CDD approach": Create a piece of code which access the required hardware and map this code to a trusted OsAppication with memory protection disabled. This allows the code to have full access. From within your BSW module all hardware access must then call this small piece of code. In this case this code has full access to hardware.
2. "Hardware approach": If possible map the hardware registers into the address space of the partition which requires the access. This normally opens the access to these registers for the BSW modules which are located in the partition. The availability of this method depends strongly on the used microcontroller and the capabilities of the memory protection unit.

Example for the "CDD approach": A CDD offers methods to read (peek) and write (poke) hardware registers. Note that in such cases it should be mentioned that additionally an access management is necessary ("Who is allowed to call these functions?") because otherwise you could not guarantee freedom from interference.

The CDD is mapped to an partition with full memory access.

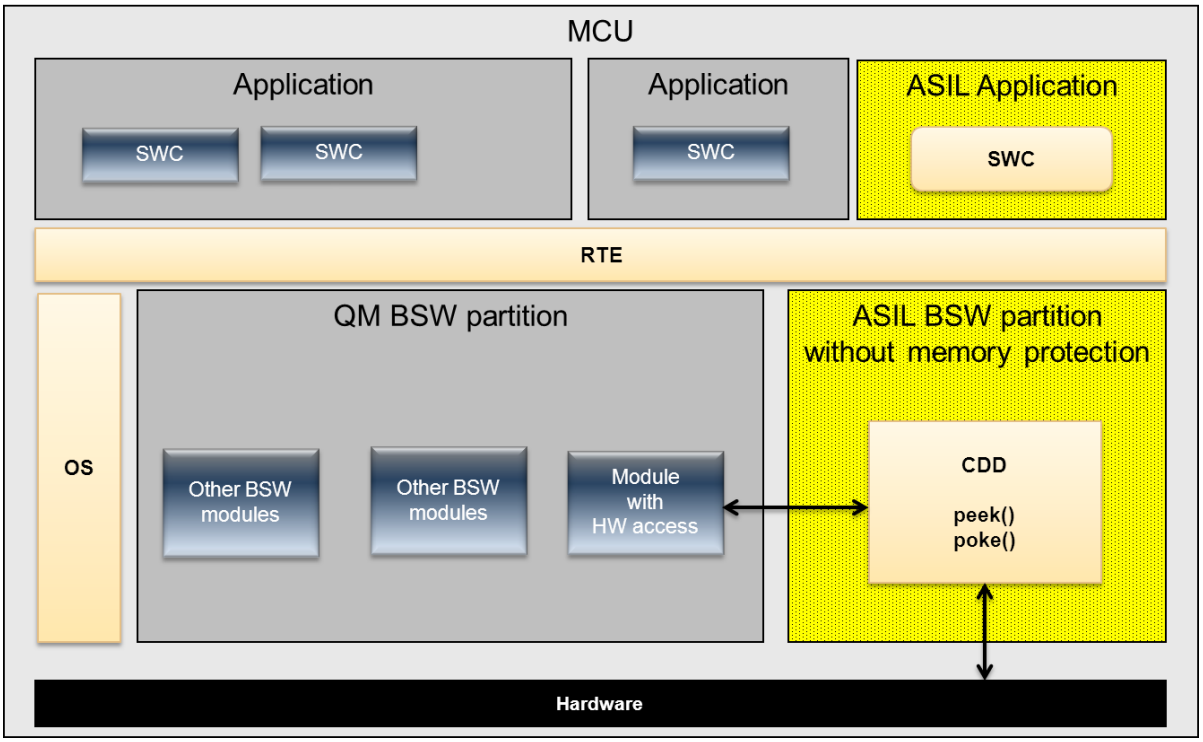


Figure 3.12: CDD approach

Note that some modules typically have implicit access, because their code is executed before the memory protection scheme is started in the Os. Details can be found in the next chapter.

3.2.5 Startup, Shutdown and Sleep/Wakeup

3.2.5.1 Startup

In AUTOSAR the startup is handled by the EcuM module. It takes care about the right order during system start. In an ASIL system the user has to take care that during startup no relevant data is overwritten or the issue is at least detected. Such faults can happen because the memory protection is not yet running because the Os is not yet started. The following figure from the EcuM shows the default sequence during startup.

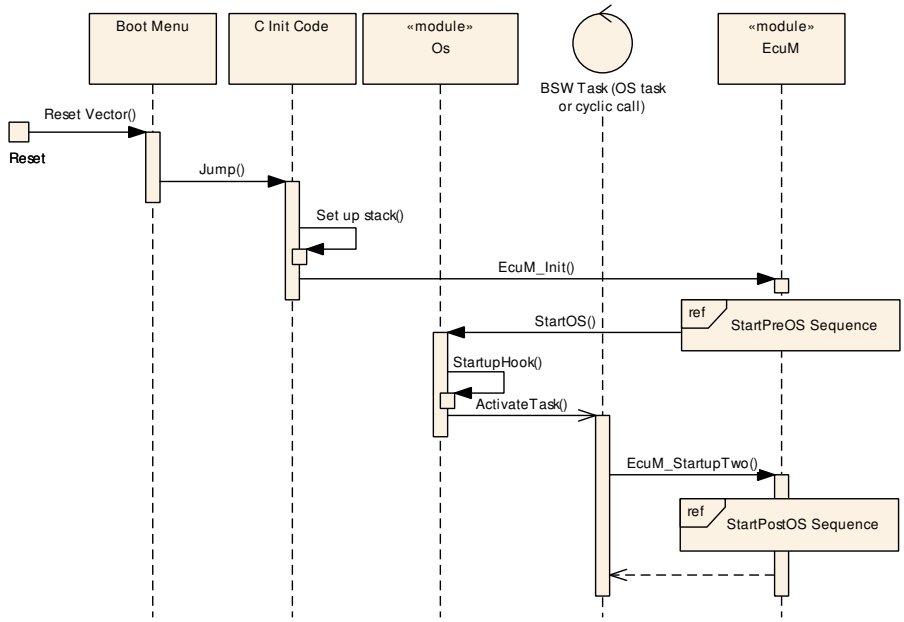


Figure 3.13: Startup of ECU

As a general hint it is always good to minimize the amount of code which is executed before the Os starts. Depending on the ASIL it might be required to develop all code of the startup as ASIL or to find other ways to make sure that nothing bad happened during startup e.g. by checking relevant data at a later point in time.

3.2.5.2 Shutdown

For the shutdown we have to distinguish different scenarios. From AUTOSAR perspective the EcuM also handles the shutdown. Compared with the startup we have a situation where the memory protection is enabled also during shutdown.

3.2.5.3 Sleep / Wakeup

In AUTOSAR EcuM takes also care for the sleep / wakeup handling. If a system has specific safety requirements in this area, also the EcuM shall take care. E.g. check if users are allowed to trigger a sleep / do a wakeup validation.

3.2.6 Error handling

When BSW modules are mapped to different partitions they do not change the overall AUTOSAR error handling. E.g. calls to Dem or Det still take place and - depending on the mapping - may cross partition boundaries.

Nevertheless the use of more than one partition with BSW modules introduces some new fault scenarios:

- A BSW function located in a trusted memory protected partition may cause a memory violation.
- A BSW function may be executed with timing protection and may run out of time, causing a timing violation.
- A BSW function may try to access some hardware registers where it has no access to.
- ...

In AUTOSAR systems without BSW distribution these issues are normally not detected because the timing protection is not used for BSW tasks. This may cause problems during normal program execution probably or at a later point in time.

In a partitioned system where the protection is enabled also for BSW modules the problems are detected and reported via the OsProtectionHook. Restarting of single partitions is not possible. This means that also for partitioned systems a protection fault is fatal and will cause a restart of the system. The advantage is that the fault can be detected much earlier and the restart can be made in a more controlled manner.

3.2.7 Timing protection

From the errors mentioned in Chapter 3.2.6 the timing faults are a special case since they may happen at any time. E.g. consider the following example:

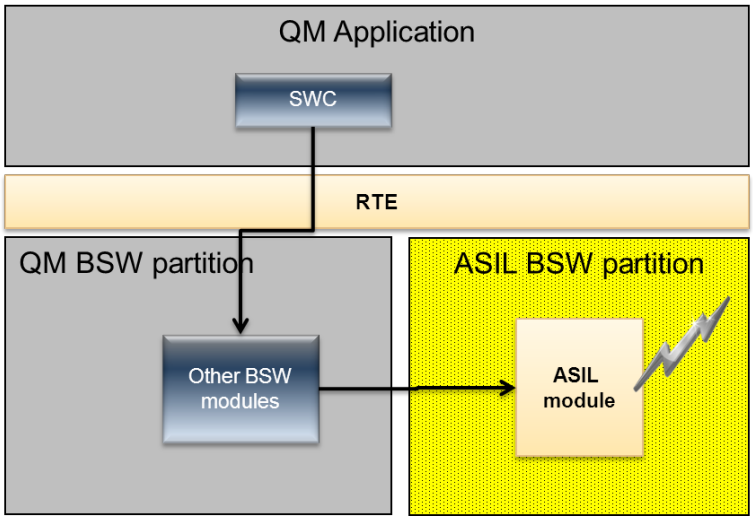


Figure 3.14: Timing fault

Here a runnable of a SWC calls an AUTOSAR service and continues execution in the QM BSW partition. From here a call to an ASIL module located in a different partition is performed. Then - right within the ASIL module - the timing violation takes place. The ASIL module has no chance to detect the problem and the system will shutdown.

To avoid such scenarios, trusted OsApplications have the ability to delay timing violation up to the point when the causing task (or ISR) leaves the partition. If both BSW partitions have the flag enabled the timing violation is reported at the point where the call from the SWC to the BSW module returns. Then it causes a violation and may end with a termination of the QM Application partition. The advantage here is that the BSW does not report the issue and there is no need for a shutdown.

The feature can be enabled for each trusted OsApplication via the configuration parameter `OsTrustedApplicationDelayTimingViolationCall`.

3.2.8 Combining Safety and Multi-Core

In case ASIL systems are implemented using a multi-core architecture, all considerations made until now for both, safety and multi-core, are valid. In a multi-core system, the BSW is assigned to core specific partitions. If safety is added, we have core specific QM and ASIL partitions. The specific multi-core configuration parameters and the specific safety configuration parameters are independent and need to be set according to the multi-core respectively safety needs.

3.2.9 Performance Considerations

The main goal for BSW distribution within safety systems is the minimized effort if only (small) parts of the system need to be developed according to ASIL. The drawback is that the protection schema causes additional overhead. The amount of time required

for the overhead depends on the project and on the mapping of the BSW modules and the frequency of interaction between the partitions.

The overhead will be minimized if ...

- ... as few as possible BSW partitions are used. Adding more partitions causes in all cases more overhead.
- ... mapping of BSW modules follows the "nearest" approach. This means that modules with a high interaction should be placed in one partition. E.g. placing the whole communication stack in on partition is much faster than splitting it up and placing e.g. the PduR in a separate partition.
- ... the number of inter partition calls is minimized. The possibilities for the user are normally limited since AUTOSAR defines the interaction between the BSW modules. Nevertheless integrator code and CDDs can be written in such way that the number of such inter partition calls is minimal.
- ... specific hardware features are supported. E.g. if there is a possibility to have more memory regions by hardware they can be utilized to avoid copying data for OUT or INOUT parameters. Note that it is not enough that the hardware offers such mechanisms; the AUTOSAR vendor must also utilize it (e.g. by supporting such features in the Os or memory mapping handling).
- ... avoid IOC calls. IOC will always do a copy of your data. Thus avoiding calls to it will increase the performance. In general try to "pull" the data instead of "push", this means the caller shall (after return of `CallTrustedFunction`) try to read the data. The buffer shall be on the callee side if possible.

3.2.10 Constraints

The approach to separate BSW modules into different partitions works, but has limitations depending on the available hardware:

- On some MCUs the access to registers is limited to specific processor modes. In such cases a peek/poke approach (see Chapter 3.2.4) is usable but consumes more time than a direct access. The amount of time spend for these functions may be fine for startup or shutdown, but not during normal operation if performed with high frequency.
- Normally only write access is limited between (BSW) partitions. Sometimes even a read access to peripheral registers has write effects (e.g. reading the buffer of received characters). In such cases also the read access may be limited.
- Sometimes the hardware does not support the use of memory protection while executing in privileged modes. In such cases it is recommended to run all partitions in non-privileged modes to use memory protection. The amount of code which requires privilege modes shall be minimal in such cases.

Note that for those measures typically the MCAL vendor is responsible. This may also apply for an MCAL qualified to an ASIL if the BSW is only QM.

4 Outlook on Upcoming AUTOSAR Versions

In this chapter, we list changes to the distribution of BSW that may occur in the next backward incompatible release of AUTOSAR. Hence, the content of this chapter is not applicable to AUTOSAR 4.x implementations, but is supposed to show possible extensions and enhancements for future versions of AUTOSAR. Note that all these topics need to be considered in parallel, because definitions of BSW functional clusters and their standardized interfaces, which will be named "Standardized AUTOSAR BSW Cluster Interface" then, are needed to support a safety use case.

4.1 Known limitations

The support for Basic Software Allocation in AUTOSAR is currently limited to backward compatible changes (w.r.t. AUTOSAR 4.0.3). This currently results in the following restrictions, which may not apply to future releases of AUTOSAR:

- Communication between master and satellites is not standardized.
- BSW functional clusters and their AUTOSAR BSW Cluster Interface are not standardized.

4.2 Inter BSW module calls in distributed BSW

Currently the BSW distribution has the constraint that existing QM modules shall be reused as is. If we would weaken this we can allow a more performant communication between modules. E.g. it could be possible to include `SchM_Call`s directly at the caller and to avoid the stubs. (Typically the caller knows the context of the call and can prepare the best environment for the call).

Also multi-core systems would benefit if all inter BSW module calls are encapsulated with a `SchM_Call`.

4.3 Standardized BSW functional clusters

BSW functional clusters are groups of functionally coherent BSW modules. Each BSW functional cluster includes a set of BSW modules. It is possible to have several functional clusters of the same type (e.g. several I/O clusters in different partitions), each using a different set of modules (e.g. IOHWA + ADC in one partition and IOHWA + ADC + DIO in the second partition). Each functional cluster has a "AUTOSAR BSW Cluster Interface", which is used to communicate with other functional clusters

BSW functional clusters can be allocated to different partitions, and functional clusters of the same type can be available in several partitions. Different functional clusters can be allocated to the same or to different partitions.

The same functional cluster can only exist at most once in each partition.

But this whole cluster allocation and the resulting real interfaces are not yet standardized, just the technique is proposed here. Thus:

Upcoming versions of AUTOSAR may standardize one or more of the following:

- Define which modules are assigned to which BSW functional cluster (=> "Standardized BSW functional cluster"). It is very likely that modules of the same stack (for instance I/O services, I/O hardware abstraction and I/O drivers) will be assigned to the same functional cluster.
- Standardize communication between functional clusters of different types via "Standardized AUTOSAR BSW cluster interfaces", as shown in Figure 4.1.

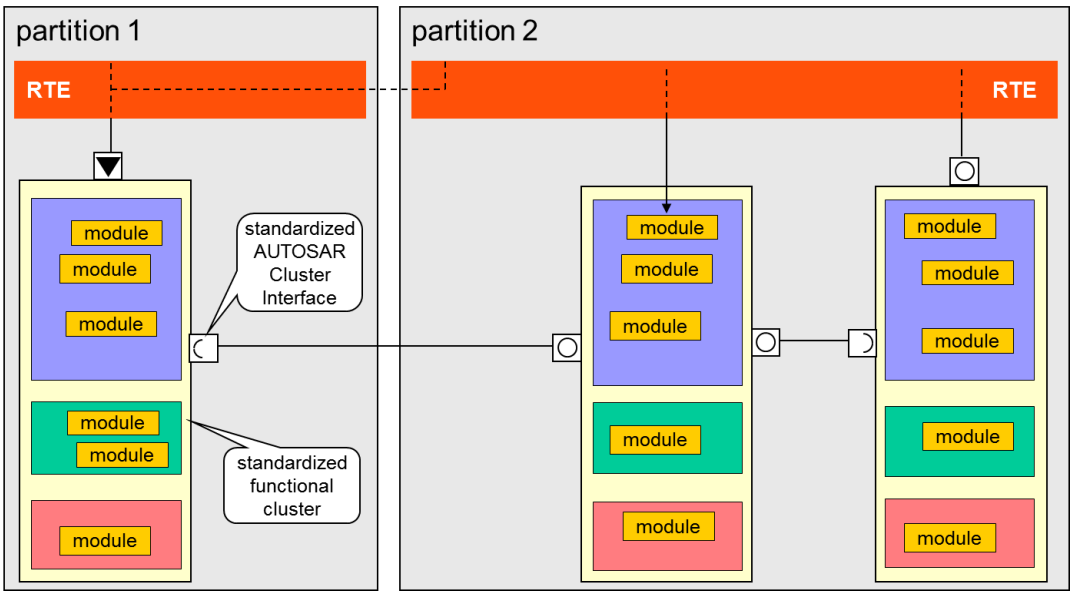


Figure 4.1: Standardized BSW Functional Clusters

5 Glossary

All technical terms used throughout this document - except the ones listed here - can be found in the official AUTOSAR glossary [4] or the Software Component Template Specification [5].

5.1 Acronyms and abbreviations

Abbreviation	Explanation
ASIL	Automotive Safety Integrity Level
QM	Quality Managed (i.e. not developed according to ASIL requirements)
IOC	Inter OS-Application communicator, part of OS
MCU	microcontroller unit, μC
MCAL	microcontroller abstraction layer

5.2 Technical Terms

Term	Explanation
BSW functional cluster	A coherent group of BSW modules. The technique is proposed in this document, but a real allocation of modules to clusters is currently not standardized. A BSW functional cluster may be similar to what usually is called a "stack", but it would also be possible to combine several stacks into a cluster or to distribute a stack across several clusters. A BSW functional cluster includes the superset of modules, which can be part of the functional cluster, but not all modules need to be available in a specific implementation. In case the real allocation of BSW modules to BSW functional clusters is standardized in future, they probably will be named "Standardized BSW functional clusters". BSW functional clusters can be allocated to different partitions, and clusters of the same type can be available in several partitions (either on the same or on different cores). Different functional clusters can be allocated to the same partition. Note: Contrary to ICC2 clustering, the internal structure and the interfaces between the modules within the functional cluster are not affected by the BSW multi-core support in AUTOSAR 4.1.1.
AUTOSAR BSW Cluster Interface	Interfaces between BSW functional clusters resulting from a vendor/project specific definition of BSW functional clusters. The technique is proposed in this document in a vendor/project specific way. But the allocation of modules to BSW functional clusters and thus the resulting interfaces are not standardized yet (if possible at all). This term may be defined in an upcoming release of AUTOSAR as "Standardized AUTOSAR BSW Cluster Interface" after standardization. Contrary to the standardized AUTOSAR interfaces, AUTOSAR BSW Cluster Interfaces shall not be connected to SW-Cs or BSW modules on other MCUs.
Master	Part of a distributed BSW module that coordinates requests by satellites and can filter or monitor incoming satellite requests. The master may work properly even if the satellites are not available. In future versions of AUTOSAR, where case partitioning may be used to enhance safety, it may be recommended or mandatory to locate the master in a partition with a high trust level, e.g. in a trusted partition.



△

Satellite	Part of a distributed BSW module. The distribution of work between master and satellite is implementation specific. One possibility is that the satellite only provides the interfaces to the other modules and routes all requests to the master and answers back to the other modules. In a different scenario, the satellite can provide the full functionality locally and only synchronizes its internal states with the master if necessary. Intermediate forms between these two scenarios are possible, but the satellites in general cannot work without the master.

6 References

[1] Specification of Memory Mapping
AUTOSAR_CP_SWS_MemoryMapping

[2] Specification of PDU Router
AUTOSAR_CP_SWS_PDURouter

[3] ISO 26262:2018 Road vehicles -– Functional Safety
https://www.iso.org

[4] Glossary
AUTOSAR_FO_TR_Glossary

[5] Software Component Template
AUTOSAR_CP_TPS_SoftwareComponentTemplate