

Multi-Core

User Manual

How to deal with MICROSAR Multi-Core Projects

Version 1.0.0

Authors	Alexander Zeeb
Status	Released

Document Information

History

Author	Date	Version	Remarks
Alexander Zeeb	2018-09-12	1.0.0	Released

Reference Documents

No.	Source	Title	Version
[1]	Vector	AN-ISC-8-1216 Synchronization of Rte Modes on Multiple Cores	1.0.0
[2]	Vector	Technical Reference MICROSAR Diagnostic Event Manager (Dem)	8.6.0

Contents

1	Introduction.....	6
2	Multi-Core Support in AUTOSAR.....	7
2.1	Terminology and Basic Concepts.....	7
2.1.1	Base Software Satellites / Proxies.....	7
2.1.1.1	Satellites.....	8
2.1.1.2	Proxies.....	9
2.1.2	Core Types.....	9
2.1.2.1	Master Core.....	9
2.1.2.2	Slave Core.....	10
2.1.2.3	BSW Core.....	10
2.1.3	Partitions.....	10
2.2	Memory.....	10
2.2.1	Flash Memory.....	11
2.2.2	RAM Memory.....	11
2.2.3	Memory Mapping.....	11
2.3	OS.....	12
3	Memory Abstraction.....	13
3.1	Memory Map Files.....	13
3.2	Linker Script.....	13
4	Integration and Startup.....	14
4.1	Startup Code.....	14
4.2	Pre-OS.....	14
4.3	Init Task Synchronisation.....	15
4.4	BSW Initialization.....	16
4.4.1	BswM.....	16
4.4.2	Dem.....	17
4.5	WdgM.....	18
4.6	Rte.....	18
4.6.1	Trusted Function Stubs.....	19
5	Shutdown.....	21
5.1	Shutdown Master-core.....	22
5.2	Shutdown Slave Core.....	23
6	Configuration.....	24
6.1	OS.....	24
6.2	Partition Mapping.....	25

6.2.1	SWC Mapping.....	25
6.2.2	Task Mapping.....	26
6.3	Dem.....	27
6.3.1	Dem Task Mapping	28
6.3.2	Dem Data Consistency	28
6.4	BswM.....	30
6.4.1	BswM Task Mapping	31
7	Multi-Core BSW-Split.....	33
7.1	Architecture.....	34
7.2	Configuration	36
7.3	Integration Example Ethernet.....	37
7.3.1	OS	37
7.3.2	PduR.....	38
7.3.3	Dem	39
7.3.4	Det.....	39
7.3.5	XCP and UdpNm.....	40
	7.3.5.1 Configuration Example UdpNm.....	40
	7.3.5.2 Implementation Example UdpNm.....	40
7.3.6	Other Modules (BswM, Sd, NvM...)	42
7.3.7	Initialization	44
7.3.8	PduR Appl Spinlocks.....	44
7.3.9	Memory Mapping	46
8	Application SWC Design Recommendations	47
8.1	Code Design	47
8.2	Inter-Core Data Flow.....	47
8.3	Software Distribution/Mapping	48
8.4	Configuration Aspects	49
9	Glossary and Abbreviations	50
9.1	Glossary	50
9.2	Abbreviations	50
10	Contact.....	51

Illustrations

Figure 2-1	AUTOSAR Architecture 4.3.0.....	7
Figure 2-2	BSW Satellites and Proxies	8
Figure 2-3	Partitions	10
Figure 2-4	Scheduling Order (AUTOSAR OS)	12
Figure 4-1	Example Dem Pre-Init.....	17
Figure 4-2	Dem_Satellite_Init	18
Figure 4-3	Example Trusted Function Call Stub.....	20
Figure 5-1	Simplified Example Shutdown Mode Machine	21
Figure 5-2	Shutdown Master Core	22
Figure 5-3	Shutdown Slave Core	23
Figure 6-1	SWC Mapping	25
Figure 6-2	EcuC Partition Mapping	26
Figure 6-3	Task Mapping	26
Figure 6-4	Task Mapping Assistant	27
Figure 6-5	Auto Solving Action DemOsApplicationRef	28
Figure 6-6	Dem Task Mapping	28
Figure 6-7	Program Logic Atomic Compare and Swap	29
Figure 6-8	Dem Spinlock Configuration	30
Figure 6-9	Dem Exclusive Area Implementation	30
Figure 6-10	BswM Partition Specific Configuration	31
Figure 6-11	BswM Spinlco Configuration	31
Figure 6-12	BswM Task Mapping	32
Figure 7-1	BSW-Split Overview	33
Figure 7-2	BSW-Split Vertical Cut Architecture.....	34
Figure 7-3	BSW-Split Architecture – Example Ethernet.....	35
Figure 7-4	BSW-Split – Example Task Mapping.....	38
Figure 7-5	BSW-Split – Example PduR BSW Module Mapping.....	38
Figure 7-6	BSW-Split – Example Dem Configuration	39
Figure 7-7	BSW-Split – Example CDD	40
Figure 7-8	BSW-Split – Example CDD Reception	41
Figure 7-9	BSW-Split – Example CDD Transmission	41
Figure 7-10	BSW-Split – Example CDD Tx Confirmation	42
Figure 7-11	BSW-Split – Example RPC Wrapper.....	43
Figure 7-12	BSW-Split – Example PduR Appl_GetSpinlock.....	45
Figure 7-13	BSW-Split – Example PduR Appl_ReleaseSpinlock.....	45
Figure 7-14	BSW-Split – Example PduR Appl_GetSpinlockInitVal	45

Tables

Table 9-1	Glossary	50
Table 9-2	Abbreviations.....	50

1 Introduction

This document is a guideline and knowledge base which helps during project setup of a MICROSAR multi-core project. It is structured in five major sections.

1. The first gives an overview of the general goal of the multi-core support as specified by AUTOSAR, the terminology and it explains the multi-core functionalities of the MICROSAR base software.
2. The second section describes the additional steps required to deal with the multi-core particularities in general system integration and for system startup and shutdown.
3. The third section shows the multi-core-specific configuration parameters of the MICROSAR base software.
4. The fourth section explains the distribution of the communication stack (BSW-split).
5. In the fifth and last section, a set of implementation and configuration recommendations for the application SWC is given, which take the particularities of the AUTOSAR stack into account.

Depending on the previous knowledge, this document can be used differently. If multi-core is a completely new topic, it is recommended to have a look at the first section (chapter 2). It gives the foundation and explains the terminology which is used in this document.

Alternatively, the section 2 to 5 (chapters 3 to 6) of this document can be used ad-hoc as additional source of information during the integration process of the ECU. Nevertheless, chapter 2.1.1.1 is recommended in any case since it describes which base software services are provided especially execution time efficient and thus provides valuable information for application deployment.

2 Multi-Core Support in AUTOSAR

This document refers to a multi-core architecture of the AUTOSAR basic software (BSW) according to version 4.3.0. In this architecture, the application SWC can be distributed arbitrarily over the available cores while most of the BSW modules remain on the BSW core. All other cores running AUTOSAR software (aka AUTOSAR cores) provide additional instances and/or satellites of BSW modules.

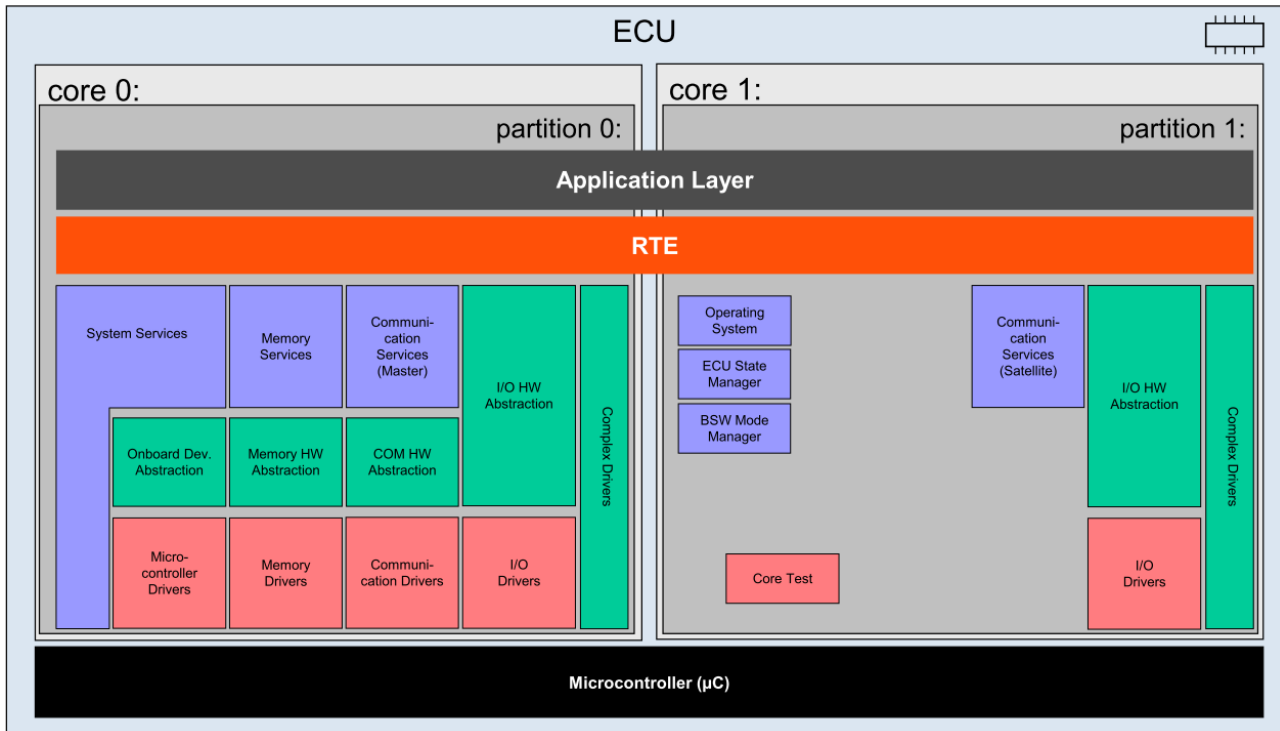


Figure 2-1 AUTOSAR Architecture 4.3.0

The Rte may span over several cores and provides transparent access to the BSW services from all cores. The Rte directly connects SWC with BSW modules respectively their satellites or, if there is no satellite, implements mechanisms for crossing the core boundary to the modules on BSW core.

2.1 Terminology and Basic Concepts

2.1.1 Base Software Satellites / Proxies

AUTOSAR describes the possibility for BSW satellites and proxies as an alternative to Remote Procedure Calls (RPC) to the server runnables. The RPC mechanism is automatically generated to connect SWCs with BSW modules located on another core. The following picture shows the basic function principle of both approaches.

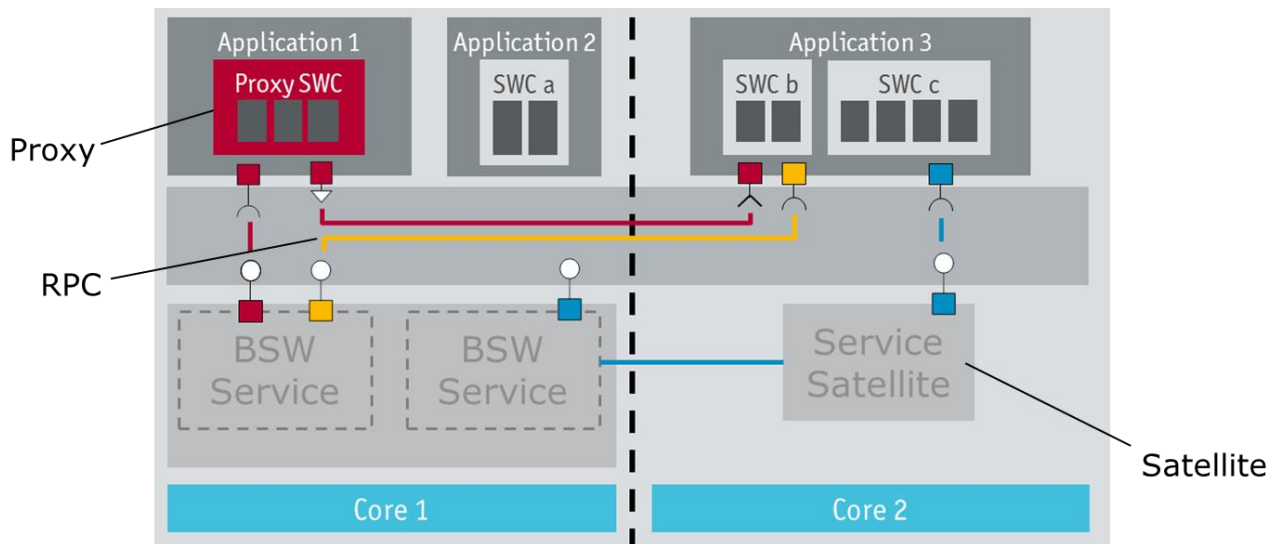


Figure 2-2 BSW Satellites and Proxies

The RPC mechanism is flexible but generates a considerable runtime overhead. The motivation of satellites and proxies, on the other hand, is to provide execution time efficient access to BSW services on all cores respectively partitions.

Satellites and proxies relax the architecture recommendation to map all SWC with BSW service needs to the BSW core. Still, even with satellites and proxies there is overhead execution time when the SWC resides on a different core than the (Master) BSW service but compared to RPC this overhead is way smaller.

2.1.1.1 Satellites

A BSW satellite is a part of a BSW service module. The idea is that the satellite offers services core-local, which means that it is executed on the same core as the accessing application SWC. The functional scope of satellites ranges from full copies of the service to very basic functionality. Common for all satellites is, that they exchange data and/or control flow with the (Master) BSW service module. Given the fact that the BSW service module can implement a tailored communication path that optimally suits to its communication needs, communication is more efficient compared to the universal path via the Rte (i.e. the RPC mechanism).

Major Benefits of Satellites are:

- > The Rte does not implement RPC for services used by SWC on different cores (improved efficiency)
- > The satellite takes over a part of the execution time (→ CPU utilization) of the BSW service since a part of the service's algorithm is executed on the caller's core

With MICROSAR R19, the Available Service Satellites which offer Interfaces Visible in the Rte are:

- > Watchdog Manager (WdgM)
- > Diagnostic Event Manager (Dem)
- > ECU Manager (EcuM)

- > Default Error Tracer (Det)

Further Satellites, Which do not Provide Services Towards the Application are:

- > Universal Measurement and Calibration (XCP)
- > Runtime Measurement (Rtm)

2.1.1.2 Proxies

A proxy translates the services of a BSW service module to a different interface type. The idea is to substitute the execution time expensive cross-core Client/Server operations with more lightweight Sender/Receiver Port Interfaces. Hence, whenever a SWC uses a service, it merely reads or writes data in shared RAM (Rte) instead of triggering a RPC.

The actual service is still called via an C/S port, but by the proxy on the same core as the service.

Proxies are located above the Rte. However, not all proxies generated by the MICROSAR stack are visible in the DaVinci Developer.

The usage of proxies requires a change of the SWC port interface from C/S to S/R. That's why a SWC must either be designed like this right from the beginning, or allow such a modification during integration.

The strategy that describes when the proxy calls the actual C/S operation of the service depends on the service itself. E.g.:

Com-Proxy

- > Transmission: Immediately before the execution of the Com main function Tx (scheduling must be configured correspondingly during integration)
- > Reception: In the context of the Com main function Rx

NvM-Proxy. Configurable:

- > Cyclic **or**
- > After change **or**
- > At shutdown
- > ...

The Available, Automatically Generated Proxies with MICROSAR R19 are:

- > Communication (Com)
- > Non-Volatile Memory Manager (NvM)

2.1.2 Core Types

2.1.2.1 Master Core

The master core of the microcontroller is started automatically after power-on. It executes its startup code (RAM, MCU initialization etc.) and finally calls the function `EcuM_Init`.

There are controllers on which more than one cores are started automatically after power-on. On these, AUTOSAR requires to emulate the master/slave startup behavior in software (compare to chapter 4.1).

2.1.2.2 Slave Core

A slave core is not started up automatically after power-on. Instead, it is started by another core during the power-on sequence. Like the master core, it executes its startup code and finally calls the function `EcuM_Init`.

2.1.2.3 BSW Core

This is the core to which the master components of the BSW modules are assigned to. However, there can be exceptions to this rule. For example, the Dem master requires to be mapped to the partition with the highest ASIL. It does not matter if this partition is on the BSW core or not.

Therefore, the following simplification can be stated for the MICROSAR stack:

> **The BSW core is the core which executes the Com or LdCom modules**

The BSW core can be the microcontrollers master core or one of its slave cores.

2.1.3 Partitions

In AUTOSAR, partitions are container which contain a set of application SWC and/or base software modules (e.g. satellites). Partitions have ASILs ranging from QM up to ASIL D and have dedicated memory protection sets.

Several partitions can share a core, thus allowing mixed ASIL software to be executed on one core.

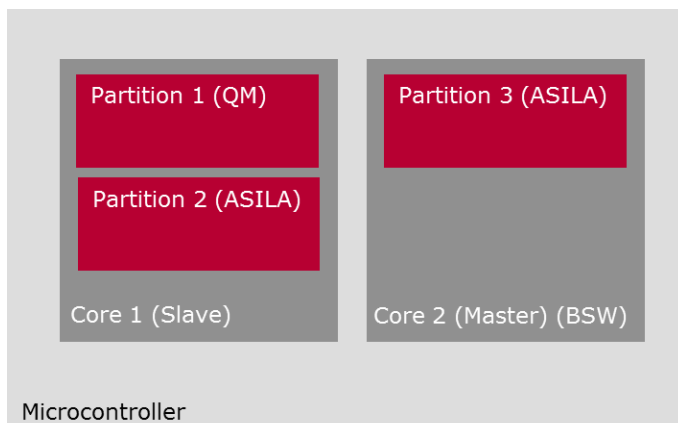


Figure 2-3 Partitions

In AUTOSAR, partitions are realized by **OS Applications**. Therefore, these two terms are typically used synonymously.

2.2 Memory

Modern microcontrollers feature several different memories. The most important types are Flash memory for persistent and RAM memory for transient program and data storage. Memory mapping in AUTOSAR mainly focuses on practical (e.g. performance, size) and safety relevant criterions.

2.2.1 Flash Memory

Flash memory typically contains BSW and application SWC code together with their constants and text symbols.

Until recently, microcontrollers only provided a single flash memory unit possible arranged in multiple flash banks. Those memory units also often had several connections to the microcontroller's interconnect, allowing dedicated, prioritized paths between memory and cores.

Newer controller architectures provide several flash memories, allowing advanced use cases for over-the-air update but also for segregation and/or performance optimization.

2.2.2 RAM Memory

RAM memory is mainly used to store variable data and the stacks. The heap memory (if used) is also kept in RAM. Another less typical use of the RAM is to store executable code. This is often done by flash bootloaders that have to execute while writing to flash memory.

Modern microcontrollers have several physical RAM units. The number and size of these units greatly depend on the microcontroller respectively its derivative. Nevertheless, RAM can in principle be categorized into:

- > **Core local RAM:** Core local RAM is as the name implies tightly connected to a processor core. Usually the core can access this memory without crossing the microcontroller's interconnect and thus suffer from less latency reducing the memory-induced wait states in the core's CPU. Despite this tight connection, other cores usually can also access core local RAM of foreign cores. This can be restricted respectively using the memory protection unit (MPU).

Core local RAM (or a part of it) is also often cached. Writing a new value to a variable may therefore get visible in RAM only after a delay (when the customer implements a cache coherence protocol or the cache controller writes back the new value).

- > **Global RAM:** Global in this context means that such RAM is not assigned to a specific microcontroller core (or peripheral). Instead, it is connected to the microcontroller's interconnect and thus equally accessible by every bus master (core, DMA). Typically, it has a higher access latency compared to core local RAM. Global RAM is often not cached, which qualifies it as shared memory for inter-core data exchange.

Microcontrollers may have several global RAM units. Each of them is connected individually to the interconnect and, depending on the microcontroller architecture, might not be accessible with the same latency from every core.

2.2.3 Memory Mapping

AUTOSAR allows a very fine-grained mapping of memory symbols to linker sections and hence addresses in the μ Cs memory. Memory symbols are basically either code, text or data. Data symbols are divided into sub groups depending on their type (constants or variables) and size. There are sub groups for each BSW module and for each SWC. Therefore, it is easily possible to map all memory symbols of a BSW module or an application SWC to specific memory sections.

Rationales for such a fine-grained mapping are:

- > **Safety:** Data is mapped to memory sections on which only trustworthy cores have write access by the MPU.

- > **Performance:** Mapping of symbols which are only or mostly used by a single core to a dedicated memory unit (RAM, flash) minimizes interference with other cores during memory access and thus reduces latency.

2.3 OS

The AUTOSAR OS is an Asymmetric Multi-Processing (AMP) system with the different OS entities statically bound to cores.

However, only OS Applications are explicitly mapped to cores during configuration. Other entities like Tasks, ISRs, Alarms and Schedule tables are referenced by the Os Applications and hence are mapped indirectly.

Task scheduling is done core-locally by the different OS instances. It is nevertheless possible to set events, start tasks etc. on different cores influencing the other core's schedule. The priority based scheduling order is core-local and regards only tasks mapped to the same core. There can be, for example, an idle task running on one core, while several higher prior tasks are waiting on another core.

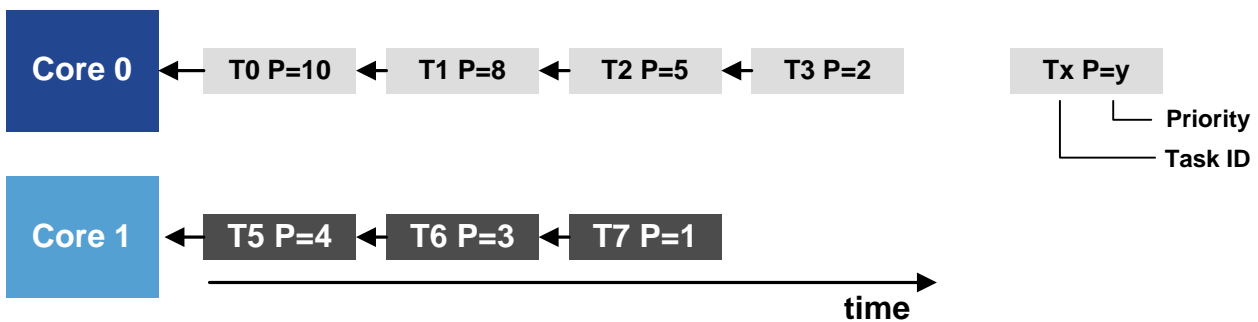


Figure 2-4 Scheduling Order (AUTOSAR OS)

For this reason, the definition of a desired program sequence by setting task priorities, is not possible for software distributed over several cores.

3 Memory Abstraction

Memory abstraction is a functionality specified by AUTOSAR and implemented by MICROSAR allowing to assign memory sections (var, code, text...) to memory areas (memory regions in physical memory). By doing so, segregation of memory due to safety and/or performance reasons is achieved.

The following multi-core BSW modules require special consideration during memory mapping in multi-core and respectively multi-partition systems. Details on the mapping requirements are given in the module's technical references.

- > Dem
- > Os
- > Rte
- > WdgIf
- > WdgM
- > Xcp

3.1 Memory Map Files

Depending on the configuration, different module-specific memory map header files need to be included in the central file `MemMap.h`.

There are only two constraints within `MemMap.h`:

- > `Rte_MemMap.h` needs to be included first
- > `Os_MemMap.h` is the last include in the include section

**Example**

```
#include "Rte_MemMap.h"
#include "Dem_MemMap.h"
/* ... */
#include "Os_MemMap.h"
```

3.2 Linker Script

The effective assignment of memory sections to memory areas (physical memory addresses) is done during linking. It is therefore necessary to create an appropriate linker script which describes all the relevant memory areas of the controller for application, BSW, OS, Peripherals (used by DMA) etc.

The linker script is highly compiler and project dependent. Hence no example can be given by this document.

4 Integration and Startup

4.1 Startup Code

The startup code is executed before a core calls the `C_main()` function. Typical tasks of the startup code are:

- > μ C configuration (PLL, memory wait states, FPU ...)
- > Register initialization
- > Base address initialization (interrupt and trap vector tables)
- > Stack pointer initialization
- > Watchdog configuration
- > Memory initialization (initialized variables and `zero init`)

On multi-core controllers, all cores need to execute a startup code. For most cases, it is advised to implement a dedicated code for each core. Otherwise, unexpected behavior is likely to occur since all cores use the same stack pointer and startup tasks like memory initialization or watchdog configuration are done by two or more cores consecutively or simultaneously.

Additionally, the startup code may have to fulfill the AUTOSAR requirement that only one core shall start up initially after power-on. All other cores are started by the BSW (respectively OS) during its startup. If the microcontroller cannot support this behavior natively, it must be emulated by the startup code. The startup code must prevent all cores except the master core from leaving the startup code until the OS was started on the master core by calling `StartOS()`

4.2 Pre-OS

The first function which is executed after the startup code is the main function. This can either be a dedicated function for each core or one common function which branches in dependence to the core context. This context can be determined with the OS API `GetCoreID()`.

An example of such an implementation can be found in the technical reference of the OS. The same document also describes which OS APIs need to be called by which core during startup.

The file `EcuM_Callout_Stubs.c` contains a working example implementation of the code required to start the OS. The example also shows how the slave cores are started by the master core of the microcontroller.

**Example**

```
FUNC(void, ECUM_CODE) EcuM_StartOS(AppModeType appMode)
{
    #if ( ECUM_NUMBER_OF_CORES > 1 )
        uint8 coreId;
        StatusType status;
        if(GetCoreID()==ECUM_CORE_ID_STARTUP)
        {
            for(coreId=0; coreId < ECUM_NUMBER_OF_CORES; coreId++)
            {
                if(coreId!=ECUM_CORE_ID_STARTUP)
                {
                    StartCore(coreId, &status);
                }
            }
        }
    #endif

    /* Start OS must be called for each core */
    StartOS(appMode);

    return;
}
```

4.3 Init Task Synchronisation

Multi-core systems require more than one `init`-tasks. Depending on the system configuration one of the two options apply:

- > If the system contains multi-partition BSW modules (e.g. Dem): One `init`-task per OS Application is required
- > Otherwise: One `init`-task per core is sufficient

If the `init`-tasks are configured with a sufficiently high priority and start up automatically, they will be executed nearly simultaneously on all cores.

However, certain APIs must be called in the correct sequence on the different cores respectively OS Applications. Therefore, wait-points are required within the `init` tasks. These wait points can be realized with the barrier-feature of the Gen7 OS or with global, non-cached variables.

Additionally, for some platforms (e.g. SPC58xx), it is required to enable the interrupt sources in the Gen7 OS manually. This is done in the core-specific `init`-tasks after calling `EcuM_StartupTwo()` by calling the OS API `Os_EnableInterruptSource()` for all relevant interrupts of the core. E.g.

```
Os_EnableInterruptSource(CanIsr_0_MB00To03, 0);
```

The symbolic identifiers of the interrupts can be found in the OS configuration (`Os_Types_Lcfg.h`).

Whether interrupts must be enabled manually is described in the technical reference of the OS.



Example

```
volatile uint32 gBarrierA = 0;
volatile uint32 gBarrierB = 0;

TASK(Init_Core_0)
{
    /* Initialization I (Core 0) */

    while( !gBarrierB )
    {
        ; /* Wait for Init_Core_1 */
    }
    gBarrierA = 1;

    /* Initialization II (Core 0) */
}

TASK(Init_Core_1)
{
    /* Initialization I (Core 1) */

    gBarrierB = 1;

    while( !gBarrierA )
    {
        ; /* Wait for Init_Core_1 */
    }

    /* Initialization II (Core 1) */
}
```



Caution

Other synchronization approaches may also be possible. They must, however, ensure that no application or BSW task is scheduled before the init tasks have completed.

4.4 BSW Initialization

This chapter describes the initialization particularities of the multi-core/multi-partition BSW modules.

4.4.1 BswM

BswM needs to be initialized by user code on all partitions except the BSW partition. Initialization is done in a partition-specific `init`-task which needs to be created manually. If there are more than one BswM instances on one core (partitioning use case), it is sufficient to initialize BswM in consecutively scheduled `init`-tasks. Initialization is done by calling the following API:

```
BswM_Init(ECUM_BSWM_CONFIG_POINTER);
```


4.4.2 Dem

The Dem requires a synchronized, multi-step initialization on all partitions.

Pre OS (Driver Init One):

- > Execution of `Dem_MasterPreInit()` on the core, the Dem master is assigned to. This API loads a preliminary configuration after which the NvM can begin to restore the Dem state from NvRAM.
- > Calls to `Dem_SatellitePreInit()` for all satellites on the core they are assigned to. After calling this API, satellites can report monitor results to the Dem.

Since this initialization step is performed before the OS is started, there is no OS Application specific execution context. It is therefore valid to pre-initialize the satellites on a per-core base. This means, the satellites of all OS applications assigned to one core can be pre-initialized by this core.

The assignment of satellite ID and OS Application can be found in the Dem gendata (`Dem_Lcfg.c`) in table: `Dem_Cfg_SatelliteInfo`.

Figure 4-1 shows a simple example with the following preconditions:

- > There is exactly one Dem satellite per core
- > The Dem satellite ID equals to the OS Application ID which in turn equals to the core ID: `GetApplicationID() == GetCoreID == Dem satellite ID`

```
/* *****  
 * EcuM_AL_DriverInitOne  
 * ***** */  
FUNC(void, ECUM_CODE) EcuM_AL_DriverInitOne(void)  
{  
    if(GetCoreID() == ECUM_CORE_ID_BSW)  
    {  
        Dem_MasterPreInit(Dem_Config_Ptr);  
    }  
    /* In this configuration, there is only one OS Application per core and the core IDs correspond to the satellite IDs */  
    Dem_SatellitePreInit( GetCoreID() );  
}
```

Figure 4-1 Example Dem Pre-Init

Partition Specific Init-Task

- > After the two steps above and after NvM has finished the restoration of the NVRAM mirror data, the Dem master is brought to full function by calling `Dem_MasterInit()`.
- > The satellites are initialized consecutively by calling `Dem_SatelliteInit()`.

In pre-OS phase, the initialization order of Dem master and the satellites depends on the startup behavior of the cores. Mapping the Dem master to the master core ensures the proper sequence since the slave cores are started after execution of `Driver Init Zero` on the master core.

Synchronization in the `init`-tasks can be achieved by applying the concept given in chapter 4.3. For the `Dem_SatelliteInit()` calls it is sufficient to execute them in consecutive scheduled `init`-tasks.

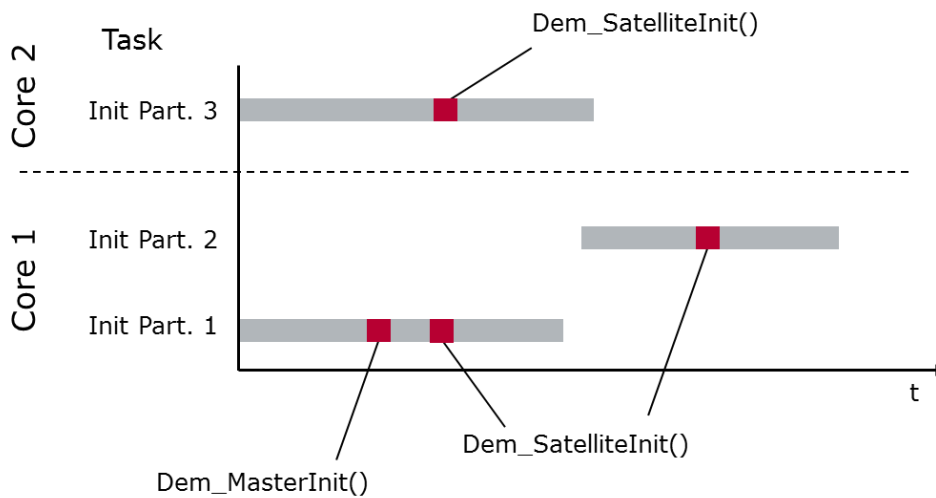


Figure 4-2 Dem_Satellite_Init

4.5 WdgM

Initialization of the WdgM is on a per-core base. It is only required on cores where the WdgM is mapped to.

For initialization, the API `WdgM_Init()` must be called after the OS has started. E.g. in the core's `init` task.

4.6 Rte

The Rte is initialized on a per-core basis. It expects a startup behavior in which the API `Rte_Start()` is called on all slave cores before it is executed on the BSW core.

The required synchronization can be realized by applying the concept described in chapter 4.3. It is recommended to call `Rte_Start()` manually also on the BSW partition instead of calling it via the BswM. Rationale is that the required call sequence can be achieved easier and more robust if all calls are done manually.

**Example**

```
TASK(Init_Task_Bsw_Core)
{
    EcuM_StartupTwo();

    /* Wait for startup of slave core (Core 1) */
    while( Rte_InitState_1 != RTE_STATE_INIT )
    {
        ;
    }

    /* Manual start of RTE. Autostart of RTE by EcuM must be disabled in configuration */
    Rte_Start();

    (void)TerminateTask();
}
```

The code above shows an example of the synchronization of the Rte start on the BSW core. The variables `Rte_InitState<core ID>` are defined in the Rte and can be used as an indicator if the corresponding Rte is initialized or not.

It is important, that these variables are mapped to a non-cached RAM area. Otherwise, the system may remain at this sync-point for an indefinite duration.

4.6.1 Trusted Function Stubs

In partitioned systems, trusted function calls are required for function calls where the caller belongs to a partition with lower ASIL than the called function.

The Rte for example uses trusted functions for client/server, mode, sender/receiver and Nv communication when the BSW runs trusted and memory from a nontrusted partition is accessed.

Trusted function calls are also used for calls between BSW modules and CDDs if they are mapped to partitions with different safety integrity levels.

If such trusted function calls are required or not depends on the system partitioning.

Trusted function calls are only supported on the same core. Cross-core calls must be implemented manually by using an IOC callback or a worker proxy mapped to a task.

The MICROSAR OS can generate stubs for the trusted functions. If, however a non-MICROSAR OS is used, the trusted function implementation needs to be done manually.

For each trusted function, the following APIs need to be implemented:

```
<ret> Os_Call_<Name Of Trusted Function>( <arg1> ... )
void TRUSTED_Os_ServiceCallee_<Name Of Trusted Function>(
TrustedFunctionIndexType FunctionIndex,
TrustedFunctionParameterRefType FunctionParams )
```

within the `OS_Call`-function the OS API `CallTrustedFunction` must be called. Arguments of this API are:

- > ID of the trusted function (from OS configuration)
- > Argument(s) of the trusted function

Within the trusted function skeleton, the actual trusted function must be called with the arguments received in the `FunctionParams`.



Example

Here is an example that has been prepared for you.

```

/*****
 * TRUSTED_Os_ServiceCallee_MyTrustedFunction
 *****/
OS_LOCAL FUNC(void, OS_CODE) TRUSTED_Os_ServiceCallee_MyTrustedFunction
(
    TrustedFunctionIndexType FunctionIndex,
    TrustedFunctionParameterRefType FunctionParams          /* PRQA S 3673 */ /* MD_Os_3673_FunctionPointer */
)
{
    P2VAR(Os_DummyTrustedFunctionPackageType, AUTOMATIC, OS_VAR_NOINIT) package;
    package = (P2VAR(Os_MyTrustedFunctionPackageType, AUTOMATIC, OS_VAR_NOINIT)) FunctionParams;

    MyTrustedFunction
    (
        package->Arg1
    );

    OS_IGNORE_UNREF_PARAM(FunctionIndex);          /* PRQA S 3112 */ /* MD_Os_3112 */
}

/*****
 * GLOBAL FUNCTIONS
 *****/
/*****
 * Os_Call_DummyTrustedFunction
 *****/
FUNC(void, OS_CODE) Os_Call_MyTrustedFunction
(
    uint8 Arg1
)
{
    Os_MyTrustedFunctionPackageType package;
    package.Arg1 = Arg1;

    (void)CallTrustedFunction(Os_ServiceCallee_MyTrustedFunction, (TrustedFunctionParameterRefType)&package);
}

```

Figure 4-3 Example Trusted Function Call Stub

5 Shutdown

If core synchronization during shutdown is required for a project, it must be achieved by means of appropriate BswM rules or user code since it is not done by the EcuM. In case there is no explicit synchronization, EcuM will shut down or reset slave cores without any further synchronization.

In a synchronized shutdown sequence, the core-local SchM instances must be de-initialized. This is achieved by calling the function `EcuM_GoDown()` on each core. The call on the master core must be done last in sequence since it ultimately shuts down the μ C.

Alternatively, the API `EcuM_GoToSelectedShutdownTarget()` can be used in combination with `EcuM_SelectShutdownTarget()` (which is BswMs default behavior) as unified entry to the EcuM. In the context of this API, EcuM calls the function suitable to the selected shutdown target:

- > `EcuM_GoDown`
- > `EcuM_GoPoll`
- > `EcuM_GoHalt`

The calls to `EcuM_GoToSelectedShutdownTarget()` or `EcuM_GoDown()` are done by the different instances of the BswM or mode machines of the Rte. To enable this, appropriate modes, rules and actions (which ultimately call the EcuM APIs) need to be configured.

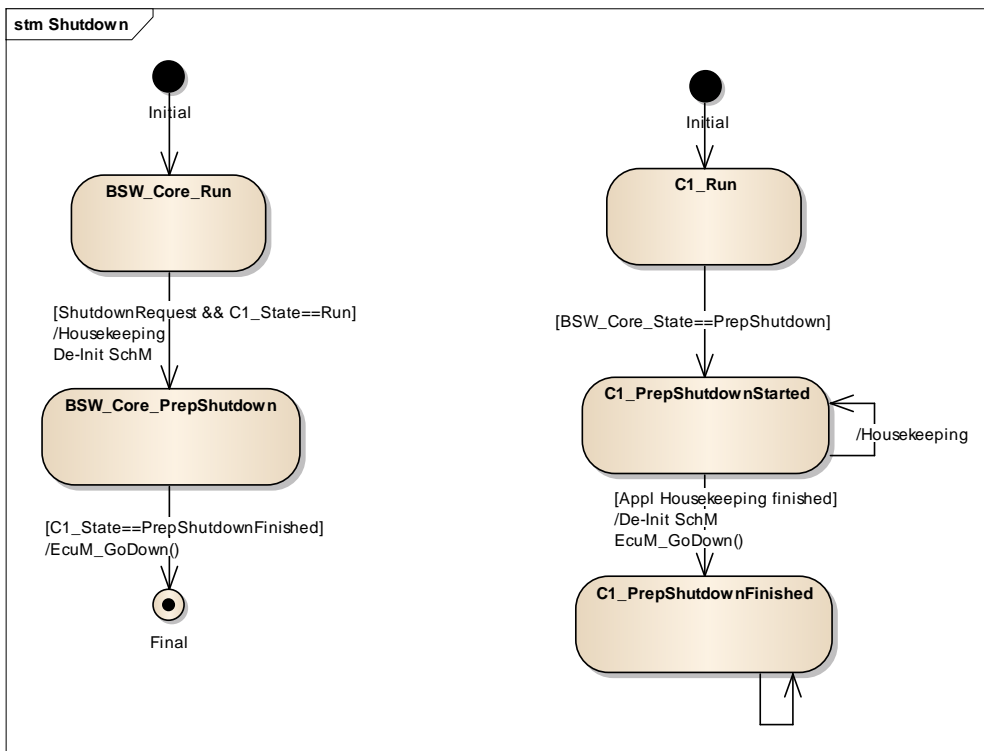


Figure 5-1 Simplified Example Shutdown Mode Machine

An example how Rte mode machines in two different partitions can be synchronized is described in [1].

Any clean-up or housekeeping of the application SW like storing intermediate data to flash memory or setting I/O pins appropriately has to be done before `EcuM_GoDown()` is called. This is typically realized with a set of suitable BswM modes (e.g. RUN, PREP-SHUTDOWN ...) which are propagated to the application SW.

From the application's point of view, the ECU can switch off any instance after `EcuM_GoDown()` has been called. It is highly recommended to keep it in a consistent and safe state (I/O, memory) until power off/reset.

5.1 Shutdown Master-core

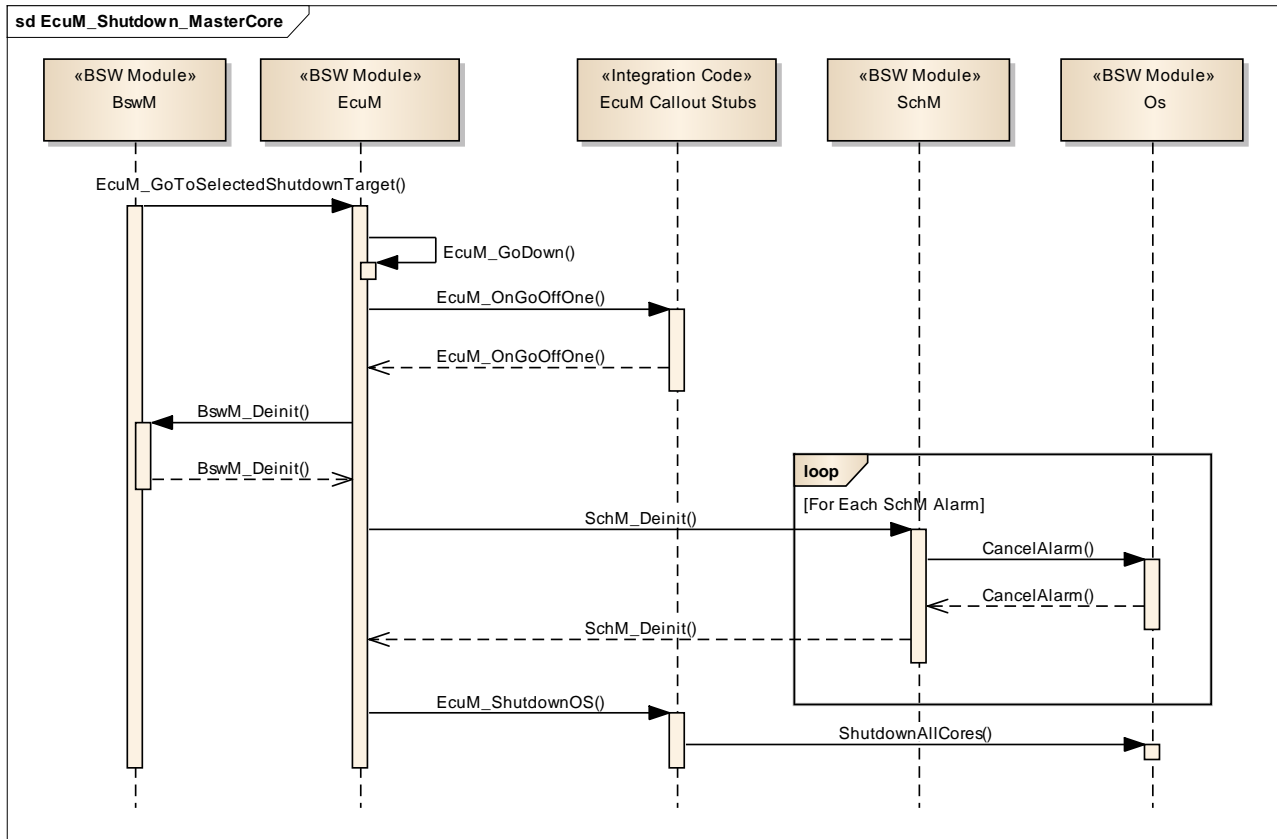


Figure 5-2 Shutdown Master Core

On the BSW core, the function `EcuM_GoDown()` de-initializes the BswM and stops scheduling of its SchM tasks by cancelling all corresponding alarms. From now on all BSW-services mapped to this core should be regarded as unavailable since their main functions are not being called any more.

As EcuM's last action, the OS API `ShutdownAllCores()` is called. It synchronizes all cores and ultimately stops the master and all slave cores.

5.2 Shutdown Slave Core

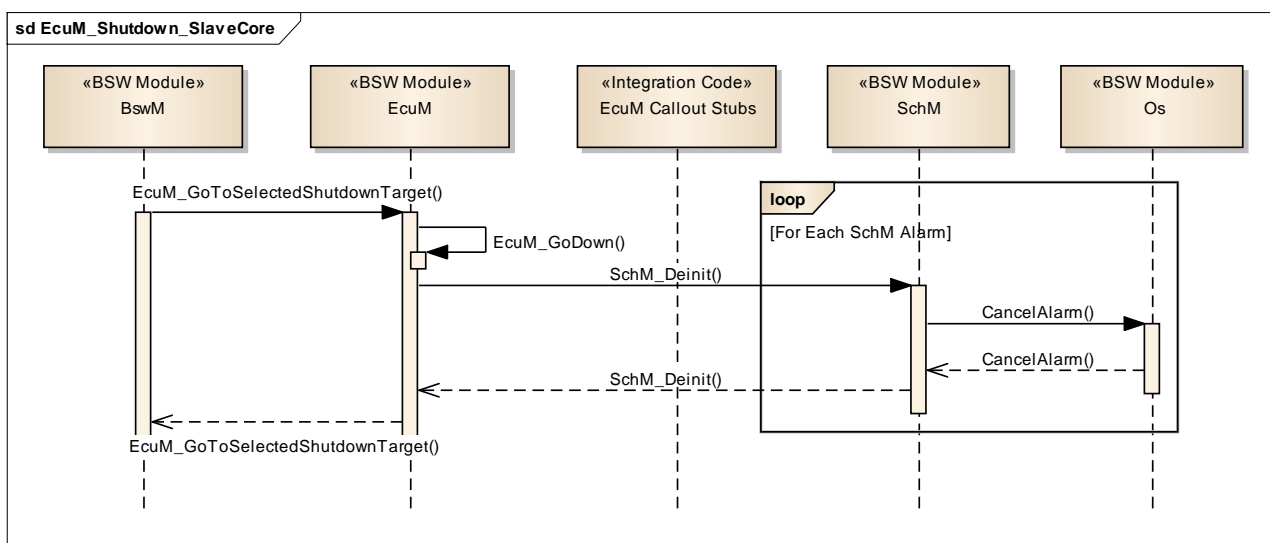


Figure 5-3 Shutdown Slave Core

On the slave core the function `EcuM_GoDown()` merely de-initializes the SchM. If there is, however, no SchM instance on the slave core; `EcuM_GoDown()` needs not be called.

6 Configuration

This chapter lists a collection of configuration parameters relevant for multi-core systems. It is not a complete list of all parameters but instead shows critical- and often overlooked parameters.

6.1 OS

- > **OsCoreIsAutosar:** Only if this check box is activated, a core can be used to execute AUTOSAR BSW or application SWC.
- > **OsCoreIsAutostart:** This must only be activated at cores which start up automatically after power on. Typically, this option is activated on only one core. If, however the master/slave core behavior is emulated in software, this option is activated on all cores which start up automatically.
- > **OsCoreXSignalChannel:** XSignal channels are required for the Os' internal cross-core communication. For example, to set an alarm or activate a task on another core. There is an auto solving action to create the required XSignalChannels. If the recommended channels are not created, the Os cannot work properly.
- > **OsCoreXSignalChannelSize:** Determines how many XSignal events can be queued for this channel. The exact number highly depends on the system configuration. A number between five and ten is however a good starting point. The Os will report an error if the queue overflowed. So, it is possible to approach the required queue size bottom up.
- > **OsIsrXSignalReceiverProvidedApis:** This list defines the Os APIs which are available via this specific XSignal channel. The Os validator constantly checks which APIs are required. Automatic population of this list is, however, only done after user interaction with the automatic solving action. It is sensible to do so before generating the Os to ensure availability of all required APIs and hence proper operation of the Os.
- > **OsIsrInterruptPriority:** Per default, the priority of the Os interrupts (timer, XSignal) is rather low. To ensure proper functionality, priority needs to be checked and potentially adjusted to be at least bigger than zero, which is equivalent to "interrupt is disabled".
- > **OsUseXSignalAsyncApiCalls:** Per default, the sender core of a XSignal waits until it was processed by the receiver core. This may however bring in considerable waiting times. To omit those waiting times, the behavior can be changed to asynchronous XSignal processing. Doing so lets the sender core return immediately after triggering of the XSignal (fire-and-forget behavior)
- > **OsSpinlockLockMethod:** Minimizing the time a core possesses a spinlock is paramount for efficient multi-core systems. Pure spinlocks do however not affect the core-local scheduling of the AUTOSAR Os. Hence it is easily possible for tasks to get preempted while holding a spinlock. To prevent this unwanted behavior, spinlocks can be combined with a locking method which affects/prevents the core-local scheduling.
- > **OsSpinlockLockType:** Minimizes the execution time of the spinlock API if set to OPTIMIZED. This is achieved by omitting the API checks specified by AUTOSAR (deadlock, nesting etc.) and keeping the spinlock data in user memory which

eliminates the need for an OS context change. However, this only applies if all accessing OS applications have the same trusted-setting.

6.2 Partition Mapping

Partition association of SWC is especially relevant when it comes to generating the interface connections in the Rte. There are different implementation variants depending on accessing an interface within the same partition or via partition boundaries.

In many cases, the Rte can determine partition association of the SWCs automatically by the task mapping of the SWC. However, there can be SWCs which only contain runnables for which task mapping is optional. A typical example for that are server runnables. In those cases, the Rte cannot determine the partition assignment automatically and the integrator needs to supplement the configuration with this information. There are two variants to do so:

- > Mapping the SWC to a partition (recommended since it causes less execution time overhead)
- > Mapping the server operation to a task (and thus implicitly to a partition). Here, the partition mapping is only a side-effect.

6.2.1 SWC Mapping

This variant explicitly links a whole SWC to a partition in the configuration. This way, the Rte has the knowledge that all unmapped runnables of this SWC belong to the referenced partition.

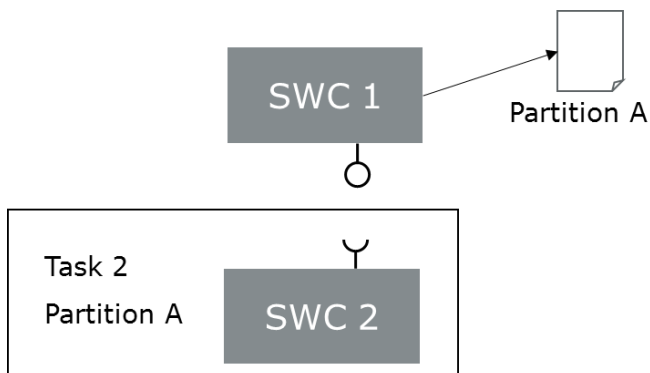


Figure 6-1 SWC Mapping

Partition mapping is done in the EcuC module in the generic editor. If a SWC shall be assigned to a partition, it must be added in the Partition Software Component Instance Ref.



Note

If caller and callee are mapped to different partitions, a task mapping for the affected server runnable is required anyway. However, if this is not the case, a direct synchronous call is possible yielding optimal performance.

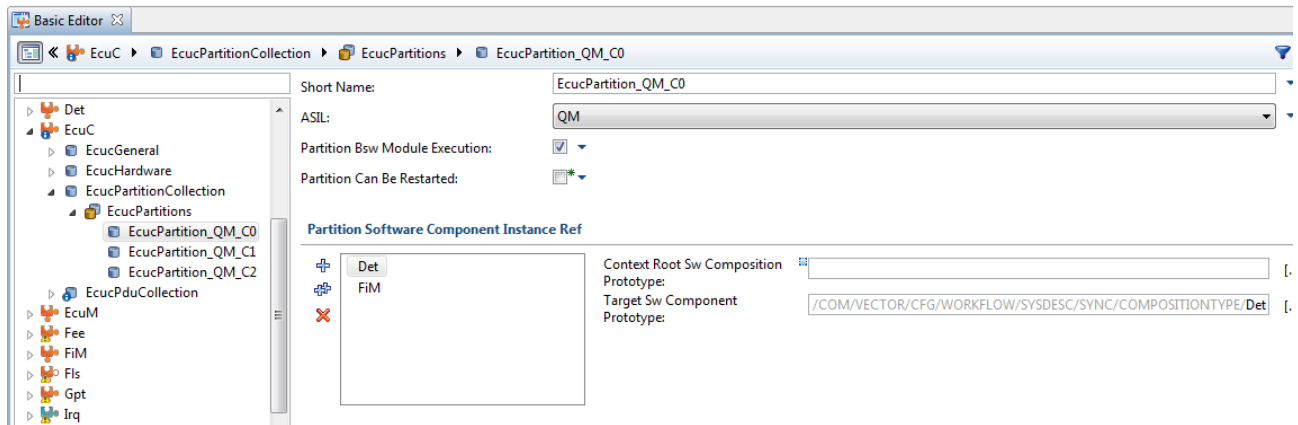


Figure 6-2 EcuC Partition Mapping

6.2.2 Task Mapping

In this variant, one or all server runnables are mapped to a task which in turn is assigned to a partition.

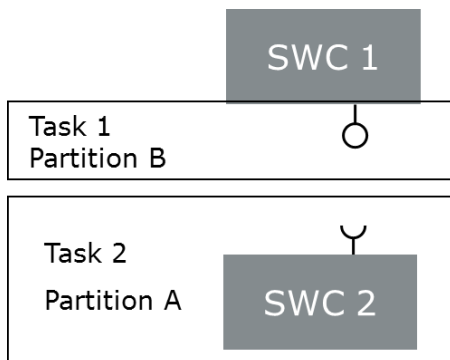


Figure 6-3 Task Mapping

The big downside of this approach is that the operation is always being executed in the context of the task it is mapped to. For many cases, this means that the Rte implements a remote procedure call (RPC) to execute the operation. This requires considerably more execution time than a plain function call which is the standard implementation if caller and callee are mapped to the same partition.

Runnable mapping is done in the Task Mapping Assistant.

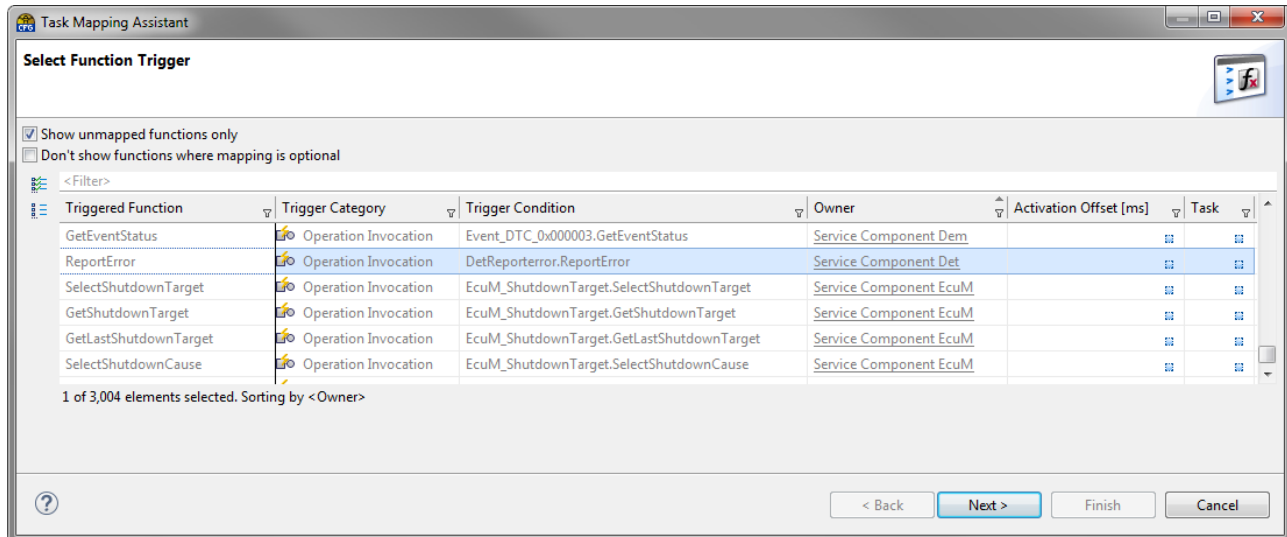


Figure 6-4 Task Mapping Assistant



Note

To avoid exceedingly long waiting times for RPC execution, runnables need to be mapped to tasks with high priority.

6.3 Dem

The advantage of the multi-partition Dem is that it provides the most commonly used service interfaces partition locally on all configured partitions. This means that the invocation of `Dem_GetEvent()` and `Dem_SetEvent()` by the application is always implemented as simple function call with little execution time.

To make use of this functionality, a couple of additional configuration steps are required.

The Dem events (`DemEventParameters`) need to be assigned to an OS application via the following parameter:

- > **DemEventOsApplicationRef:** This reference determines the Dem satellite which monitors this event. Per default this reference is not set which means that the event is handled by the Dem satellite residing on the same OS Application as the master Dem component. Creating a reference assigns the event to the satellite on the corresponding OS Application. If the OS Application was referenced for the first time, the Dem satellite is generated automatically.

Using the Dem monitor interface for an event mapped to a different OS application will result in the implementation of a RPC.

Furthermore, in the general settings:

- > **DemMasterOsApplicationRef:** OS application on which the Dem master component is being executed. This must be the OS Application with the highest ASIL.
- > **DemOsApplicationRef:** In this list, all OS applications which execute a Dem component (master or satellite) must be listed.



Note

There is an auto solving action for populating this list.








Validation  Element Usage  Find 	
9 messages in 7 categories	
ID	Message
 Cfg00020	Deviation from initial configuration (3 messages)
 DEM51137	Reference target Os/OsApplication is implicitly added to DemGeneral/DemOsApplicationRef. (1 message)
 DEM51137	<p>OsApplication TpTestApplication_OsCore0 target of event "AutoCreatedDemEvent_FlexRay_FRIF_E_SW_CH_A" is missing in the global DemGeneral/DemOsApplicationRef list - therefore implicitly added there.</p> <p> Create reference DemOsApplicationRef and use "TpTestApplication_OsCore0" as its target value</p> <p><code>/ActiveFcuC/Dem/DemConfoSet/AutoCreatedDemEvent_FlexRay_FRIF_E_SW_CH_A(DemEventOsApplicationRef)</code></p>

Figure 6-5 Auto Solving Action DemOsApplicationRef

6.3.1 Dem Task Mapping

For each Dem satellite and the master component, the cyclic main functions need to be mapped to an appropriate BSW task. The task must be assigned to the OS application, which is indicated by the satellite's name.

[illegible]

Figure 6-6 Dem Task Mapping

6.3.2 Dem Data Consistency

Dem protects the data which is being accessed from two or more cores. For this, Dem provides two alternatives.

- > **Default** – Protection via Dem's Exclusive Area 3
- > **Optimized** – Protection via a user implemented compare and swap

For best execution time efficiency with least overhead it is highly recommended to use compare and swap. To use this variant, the following switch in Dem's general settings must be set:

> DemUserDefinedCompareAndSwap

Furthermore, a function with the following signature must be provided:

```
boolean ApplDem_SyncCompareAndSwap (  
    volatile unsigned int* AddressPtr,  
    unsigned int OldValue,  
    unsigned int NewValue  
)
```

The function must realize the following program logic atomically:

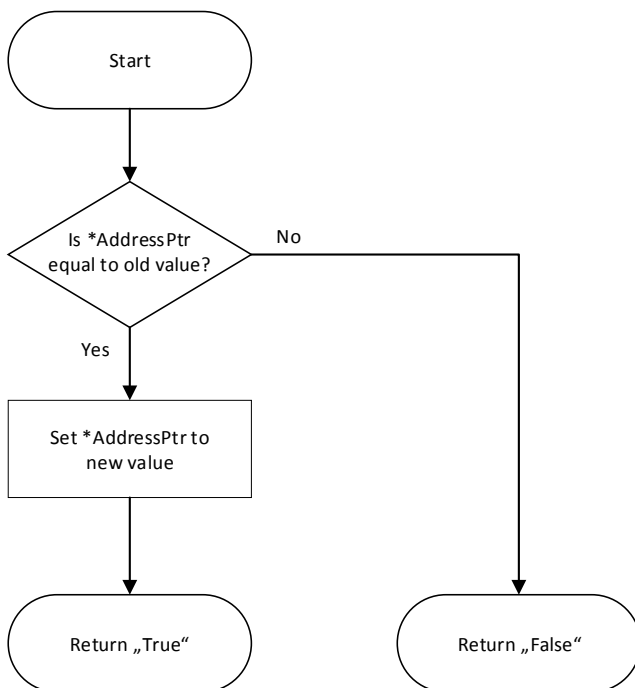


Figure 6-7 Program Logic Atomic Compare and Swap



Reference

More details on this variant are described in Dem's technical reference chapter: **4.4.1 Atomic Compare/Exchange [2]**.

The alternative variant which uses Dem's Exclusive Area 3 instead is active per default. However, Exclusive Area 3 per default has only a core-wide scope. To overcome this, it must be supplemented by a spinlock.

This cannot be done automatically. Instead, the Exclusive Area must be implemented manually. This is done by setting the **RteExclusiveAreaImplMechanism** to **Custom**.

Furthermore, a spinlock must be configured. To also prevent data inconsistencies from the same core, the **Spinlock Lock Method** must be set appropriately. E.g. LOCK_CAT2_INTERRUPTS:

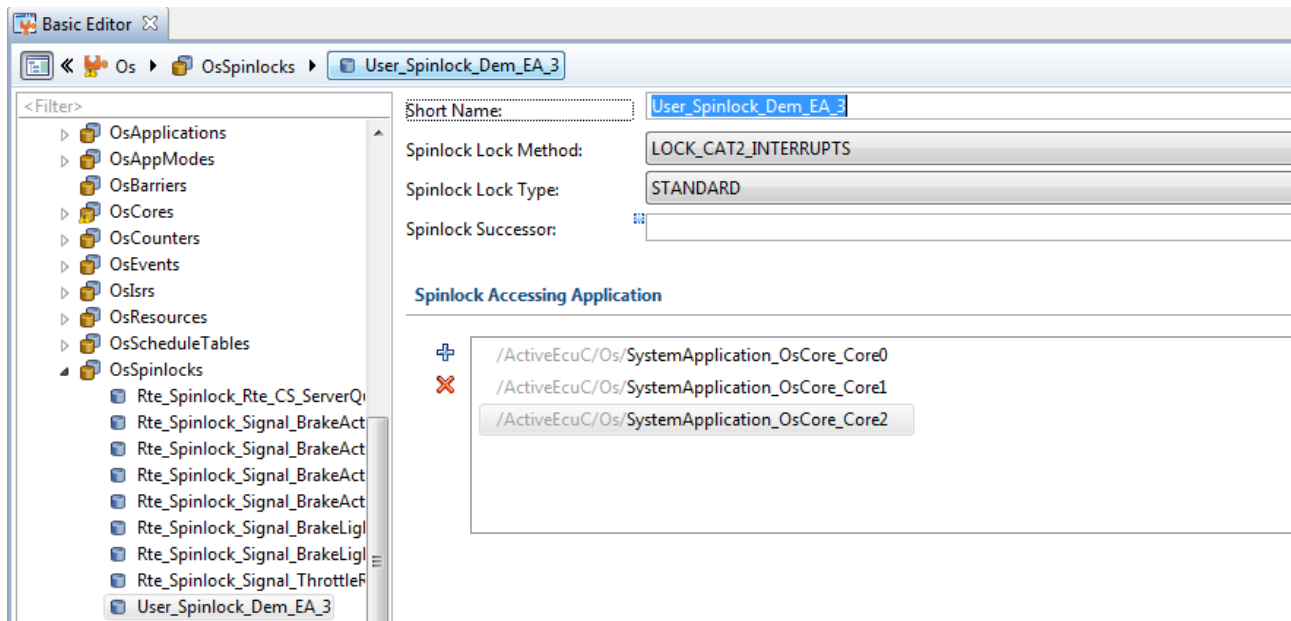


Figure 6-8 Dem Spinlock Configuration

In the implementation of the Exclusive Area, the spinlock is acquired respectively released:

```
void SchM_Enter_Dem_DEM_EXCLUSIVE_AREA_3()
{
    GetSpinlock(User_Spinlock_Dem_EA_3);
}

void SchM_Exit_Dem_DEM_EXCLUSIVE_AREA_3()
{
    ReleaseSpinlock(User_Spinlock_Dem_EA_3);
}
```

Figure 6-9 Dem Exclusive Area Implementation

6.4 BswM

BswM can provide an instance on every partition. These instances must be configured individually in the **BSW Management** view. For each partition, a new tab is created.

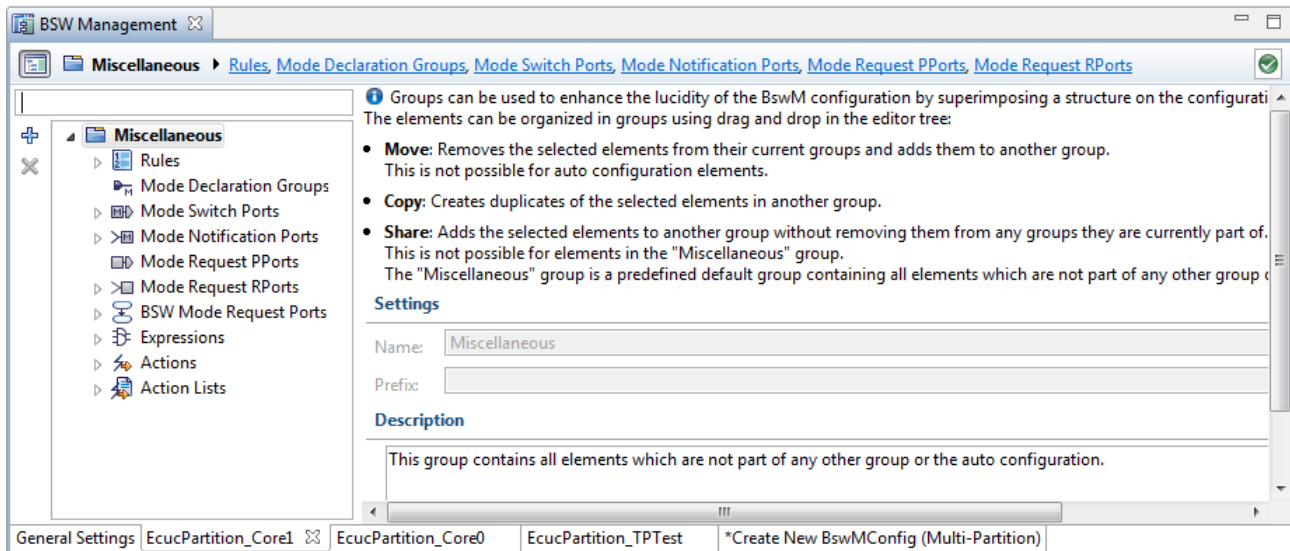


Figure 6-10 BswM Partition Specific Configuration

In addition to the partition-specific configuration, the exclusive area of the BswM needs to be extended. BSWM_EXCLUSIVE_AREA must be set to **Custom** and implemented manually. Instead of the single core mechanism like **All Interrupt Blocking**, a spinlock needs to be requested. This spinlock must be configured in the OS. The **Spinlock Lock Method** must be set to Lock CAT2 or Lock All Interrupts to provide exclusive access also on the same core context.

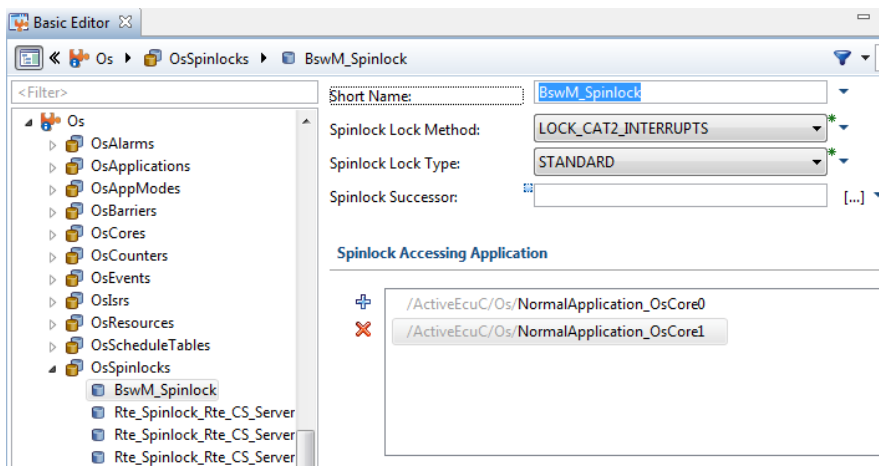


Figure 6-11 BswM Spinlock Configuration

6.4.1 BswM Task Mapping

Task mapping of a BswM instance is required if its configuration contains at least one of the following items:

- > A deferred mode request port
- > A timer
- > A Rte mode switch action

Task Mapping

Use the [OS Configurations Editor](#) to create tasks.

<Filter>

Schedulable Entities

- BswM_MainFunction
- BswM_MainFunction
- BswM_MainFunction
- Can_MainFunction_BusOff
- Can_MainFunction_Mode
- Can_MainFunction_Wakeup
- CanNm_MainFunction
- CanSM_MainFunction
- CanTp_MainFunction
- CanXcp_MainFunction
- Com_MainFunctionRx
- Com_MainFunctionTx
- ComM_MainFunction_0
- ComM_MainFunction_1
- ComM_MainFunction_2

<Filter>

Position	OsCore0	NormalApplication_OsCore0	NormalApplication_OsCore0
		SchM_Task_NormalApplication_OsCore0	Swc_Task_OsCore0
		Priority 6	Priority 25
0	BswM_MainFunction	CtApCanLinFr_core0_Init	
1	ComM_MainFunction_0	CtApCanLinFr_core0_1ms	
2	ComM_MainFunction_1	CtApCanLinFr_core0_10ms	
3	ComM_MainFunction_2	CtApCanLinFr_core0_Counter	
4	CanSM_MainFunction	CtApCanLinFr_Trigger_00	
5	CanNm_MainFunction		
6	CanTp_MainFunction		
7	CanXcp_MainFunction		
		Periodical/1 ms, Periodical/5 ms, Periodical/10 ms, Periodical/20 ms	External Trigger Occurred, Init, Periodical/1 ms, Periodical/10 ms

Figure 6-12 BswM Task Mapping

7 Multi-Core BSW-Split

The feature Multi-core BSW-Split allows core mapping of communication busses like Ethernet, CAN, FlexRay and Lin. The mapping of individual channels is not supported.

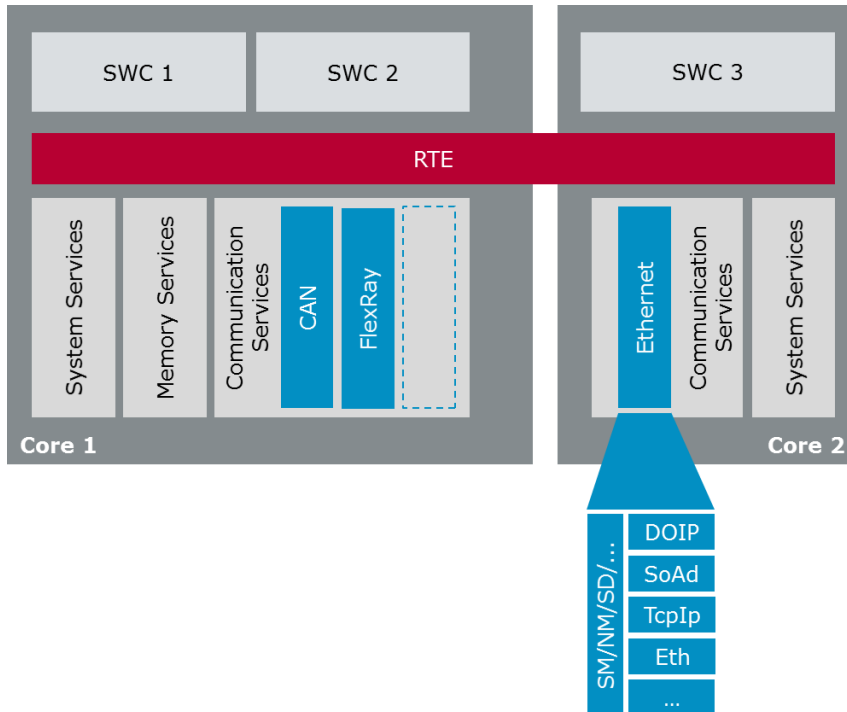


Figure 7-1 BSW-Split Overview

Mapping of a communication bus means that the execution of the cyclic main functions and the interrupts related to this bus is permanently relocated to a different core than the main BSW core.

The only motivation of such an architecture is to reduce the CPU load on the main BSW core by shifting a part of it to a different core. This load reduction is an additional degree of freedom for the multi-core software distribution beside the application SWC mapping.

However, in contrast to application software distribution, this feature is more complex and has some drawbacks. Foremost these are:

- > It does **not** increase the processing speed of the BSW (multi-core speedup) since the processing tasks are merely relocated but not parallelized. Hence, processing will even in an ideal scenario take at least the same time like on single-core systems. Realistically, it takes even longer since additional program logic (overhead) is required to exchange the data and control flow between the cores.
- > It does **not** improve efficiency of the BSW stack. Several interfaces will be across core boundaries and hence require a more complex implementation with greater execution time overhead. The BSW stack has least overall execution time when it is being executed on only one core.

- > More memory is required. The PduR buffers all PDUs which are routed between the cores to ensure data consistency and availability. This may not be required in single core configurations.
- > The extent of the load distribution depends on many factors, hence reliable estimations cannot be given. Main influencing factors are
 - > Bus type
 - > Amount and type of the PDU routings

**Note**

The best load distribution can be expected when bus systems are relocated, which cause considerable execution time in their main functions and interrupts and which transport mostly unbuffered gateway PDUs and forwardings with direct data provision.

As consequence, the data provision should be changed to “Direct” whenever possible.

7.1 Architecture

Main element of the BSW-Split feature is the enhanced PDU router. It is able to route between upper and lower layer modules on different cores.

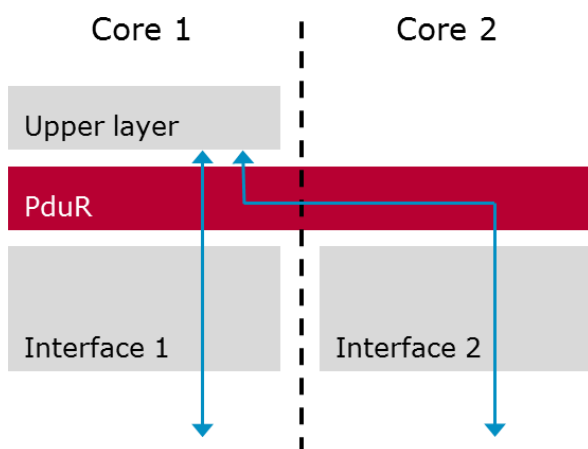


Figure 7-2 BSW-Split Vertical Cut Architecture

This multi-core routing can be applied to the following elements for the interface and transport protocol APIs:

- > Transmit requests (PDUs)
- > Rx Indications
- > Tx Confirmations

Each core handles one or more communication interfaces. If several busses are mapped to the same core, routing between these busses is performed core local.

The data exchange of the routing is done via two fundamental different approaches.

- > IF API forwardings and unbuffered routings are handled by universal, unidirectional and spinlock-free queues. The queues buffer PDUs, Rx Indications and Tx Confirmations in a FiFo manner. The reception is handled in the context of the PduR's cyclic main function.
- > TP API forwardings, buffered routings (TP and IF) and routings with trigger transmit data provision are handled by the standard PduR buffer mechanisms. The PduR employs spinlocks to ensure data consistency.

In both cases, all inter-core routings are handled **Deferred** which means that on the receiver core they are processed in the context of the PduR main function.

Non-PDU related interfaces between BSW modules on the different cores are handled depending on their multi-core capabilities. Currently only a subset of the affected modules is multi-core capable. For the other modules, workarounds are required.

- > Dem and Det are multi-core capable. Hence only their configuration must be adapted.
- > XCP and NM interfaces to the bus interfaces are handled by workaround-CDD components which route them via the PduR.
- > Modules like BswM, ComM, NvM etc. use workaround-wrapper components which decouple the API calls via the OS.

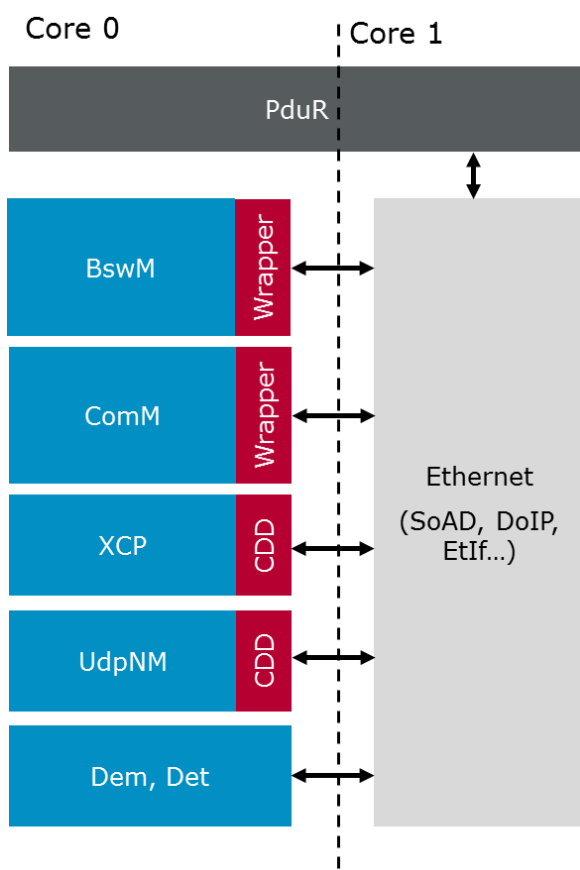


Figure 7-3 BSW-Split Architecture – Example Ethernet

7.2 Configuration

This chapter describes the parameters relevant for the Multi-core BSW-Split feature. All parameters belong to the PduR.

- > **PduRSupportMulticore:** This check box enables the general multi-core support of the PduR. It must be enabled in BSW-Split configurations.
- > **PduRDeferredEventCacheSize:** Deferred event cache allows the PduR to efficiently process the multi-core routing paths. The PduR uses this event cache to store which of the routing paths need to be handled in the next main function cycle. An event corresponds to a queued PDU, a Tx Confirmation or a Rx Indication. It is essential to configure a reasonable queue size to avoid falling back to a slower polling algorithm. The size can be estimated by determining the maximum of:
 - > The number of Tx PDUs the upper layer modules (Com, LdCom...) can transmit plus the number of gateway PDUs received on the BSW core in one main function cycle of the PduR
 - > The number of PDUs which can be received on the bus-core and which must be routed to the BSW core in one main function cycle of the PduR
- > **PduREnableDeferredReception:** Must be enabled if the relocated bus receives PDUs which are routed to the BSW core.
- > **PduREnableDeferredTransmission:** Must be enabled if the relocated bus transmits PDUs which origin from the BSW core.
- > **PduRMainFunctionPeriod:** Cycle time of the PduR main function. Lower values result in less delay during deferred reception/transmission but at the same time to more execution time overhead.
- > **PduRMulticoreQueueSize:** Size of the unidirectional PduR queues in bytes. The size can be estimated by determining the number and size of the unbuffered, direct Rx and Tx PDUs which have to be exchanged between the cores within one main function cycle.
- > **PduR_QueueOverflowNotification:** If enabled, PduR calls the function. `ErrorNotificationQueueOverflow` if a queue overflow occurs. This function must be implemented manually.
- > **PduRLock:** These are the locks used by the PduR to ensure data consistency during routing. Depending on the type of the routing path (inter core or intra core) either a classical Exclusive Area is sufficient or a spinlock is required. These locks are configured automatically and should not be modified.
- > **PduRLockRef:** Reference to the lock of the appropriate type for this routing path. This reference is configured automatically and should not be modified.
- > **PduRMulticoreRoutingPath:** Determines if the routing path is inter-core or intra-core. Must be enabled for inter core routings.
- > **PduRDestPduDataProcessing:** Determines if the routing path is either immediate or deferred. Must be set to deferred for multi-core routing paths.

- > **PduROsApplicationRef:** Defines to which OS application an upper or lower layer module belongs. In multi-core configurations, this parameter must be configured for all PduR BSW modules.

7.3 Integration Example Ethernet

This chapter describes the integration of the BSW-Split feature for a hypothetical configuration. In this example, the Ethernet bus is relocated. The integration of other bus systems can be done accordingly.



Caution

The steps and action shown in this chapter shall merely illustrate the general procedure. They **must** be adapted to the concrete project.

7.3.1 OS

In the OS, the actual relocation of the bus is done. All main functions and interrupt service routines of the Ethernet related modules (except UdpNm) are mapped to an appropriate schedule manager task allocated to the bus-core (target core). For this example, the relevant modules are:

- > Eth
- > EthIf
- > EthSM
- > Sd
- > TcpIp
- > DoIP

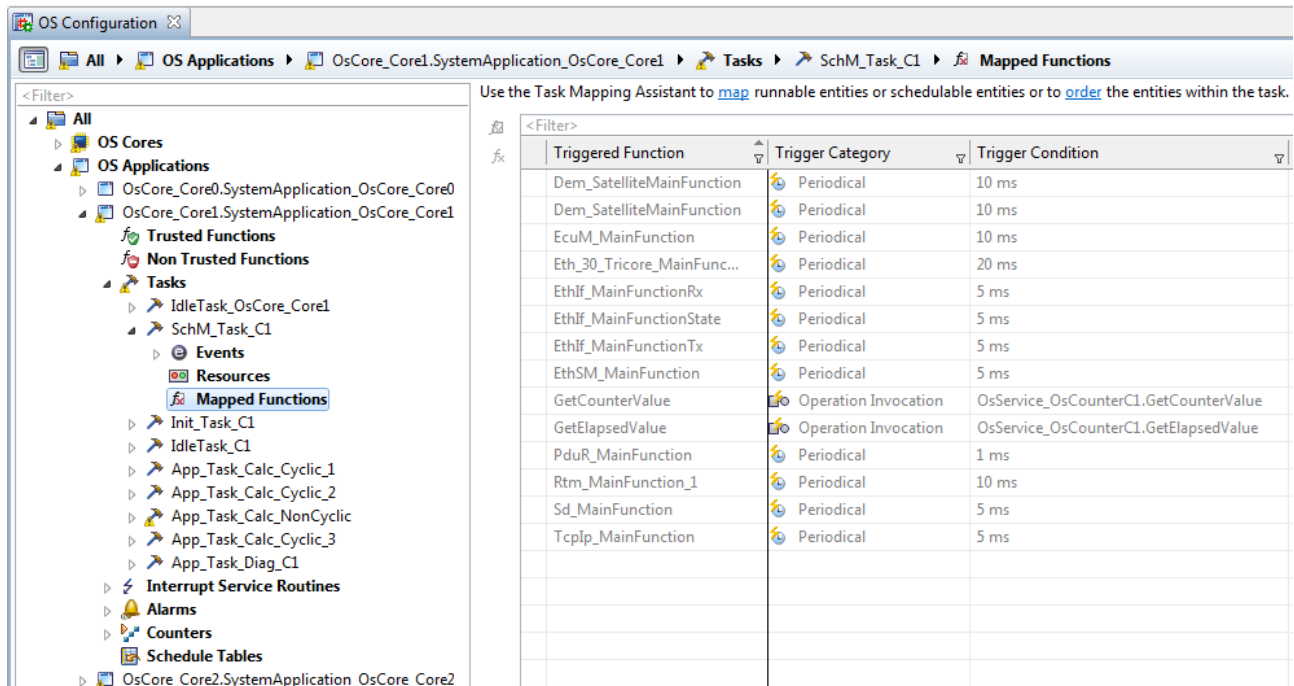


Figure 7-4 BSW-Split – Example Task Mapping

7.3.2 PduR

In the PduR the following parameters are set manually:

- > PduRSupportMulticore: **enabled**
- > PduRDeferredEventCacheSize: **128**
- > PduRMulticoreQueueSize: **4096**
- > PduRMainFunctionPeriod: **1**

The PduR BSW modules must be mapped so that the Socket Adaptor is assigned to the bus-core whereas the other upper layer modules and interfaces are mapped to the BSW-core.

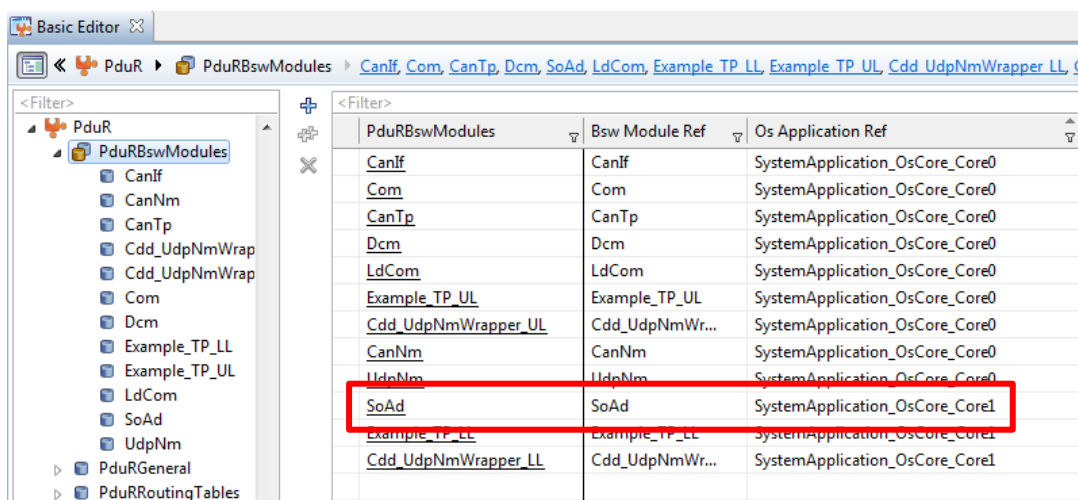


Figure 7-5 BSW-Split – Example PduR BSW Module Mapping

All other parameters can be set via the PduR autosolving actions. This includes:

- > Creation of the PduR Locks
- > Activation of deferred transmission and reception in PduR general
- > Configuration of the routing paths
 - > Multi-core routing path
 - > Lock reference
 - > Deferred processing

7.3.3 Dem

Dem is multi-core capable but requires an appropriate configuration. Dem events must be mapped to the same OS application as their diagnostic monitor. This is done by setting the application references (DemEventOsApplicationRef) of the Ethernet related modules (EthSM, Eth, Tcplp, UdpNm...) to the OS application in which the Ethernet stack shall be executed.

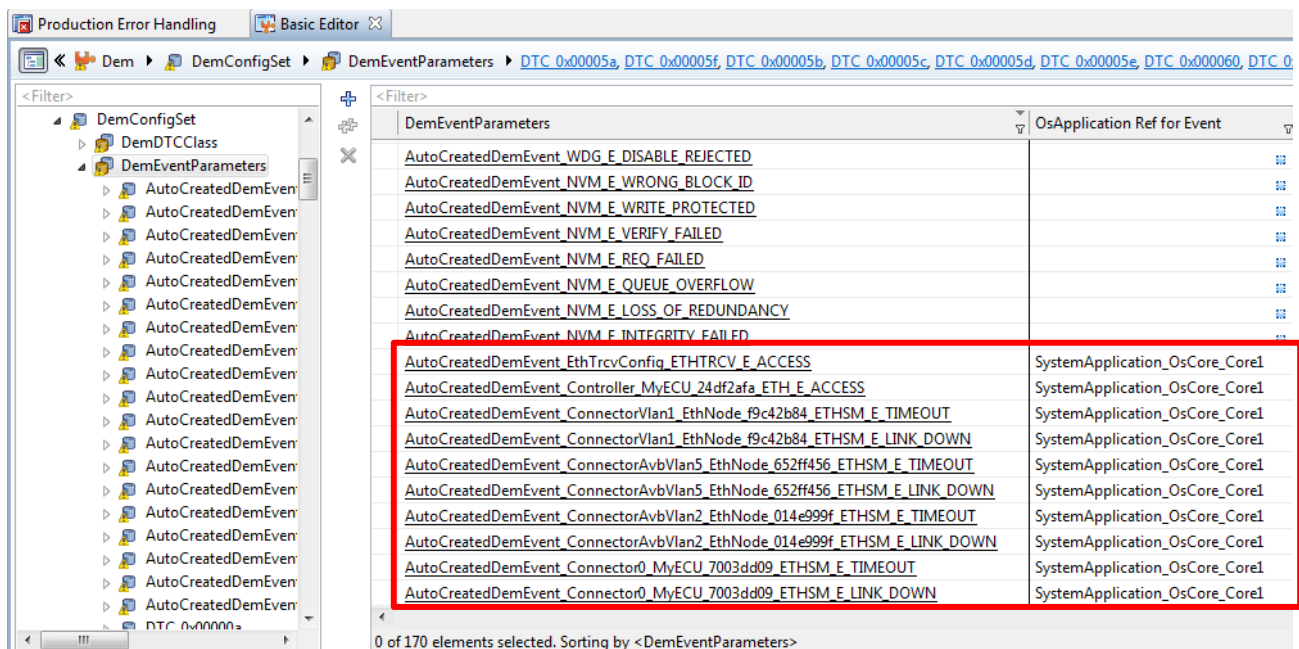


Figure 7-6 BSW-Split – Example Dem Configuration

7.3.4 Det

Det is a multi-core capable module. There is no specific switch to enable multi-core but its exclusive area must employ a spinlock in addition.

This is done by a manual implementation of the exclusive area which requests and releases the spinlock. To use a manual implementation, the parameter of **RteExclusiveAreaImplMechanism** of the DET_EXCLUSIVE_AREA_0 must be set to CUSTOM. Details on how to set this up can be found in chapter 6.3.2.

7.3.5 XCP and UdpNm



Workaround

This chapter describes one workaround. It is the decision of the integrator to choose this workaround or devise an alternative.

XCP and UdpNm are not multi-core capable but use APIs of the Ethernet stack respectively offer services which are being used by the Ethernet stack.

This is solved by implementing sets of Complex Device Drivers for both modules which redirect the affected APIs via the PduR.

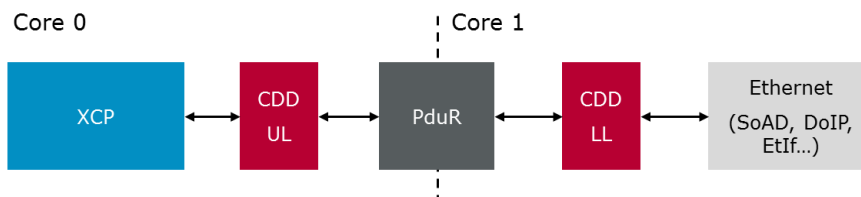


Figure 7-7 BSW-Split – Example CDD

7.3.5.1 Configuration Example UdpNm

This chapter shows the configuration steps for the CDDs required by UdpNm. For XCP a corresponding configuration is required.

- > For UdpNm the following two CDDs are created:
 - > Cdd_UdpNmWrapper_LL: Com stack contribution: CddPduRLowerLayerContribution
 - > Cdd_UdpNmWrapper_UL: Com stack contribution: CddPduRUpperLayerContribution
- > In the next step two PDUs need to be created and referenced by the CDDs. One Tx and one Rx PDU.
- > New PDU routing paths for these PDUs are created by the PduR solving actions
- > In the PduR Cdd_UdpNmWrapper_LL is assigned to the OS Application on the bus-core and Cdd_UdpNmWrapper_UL to the OS Application on the BSW core.

7.3.5.2 Implementation Example UdpNm

To redirect the APIs via the PduR, function calls in UdpNm and SoAd must be redefined to the CDD by preprocessor defines.

The actual CDD implementation uses the PduR APIs to send and receive PDUs. The implementation must realize the following sequences

- > Reception

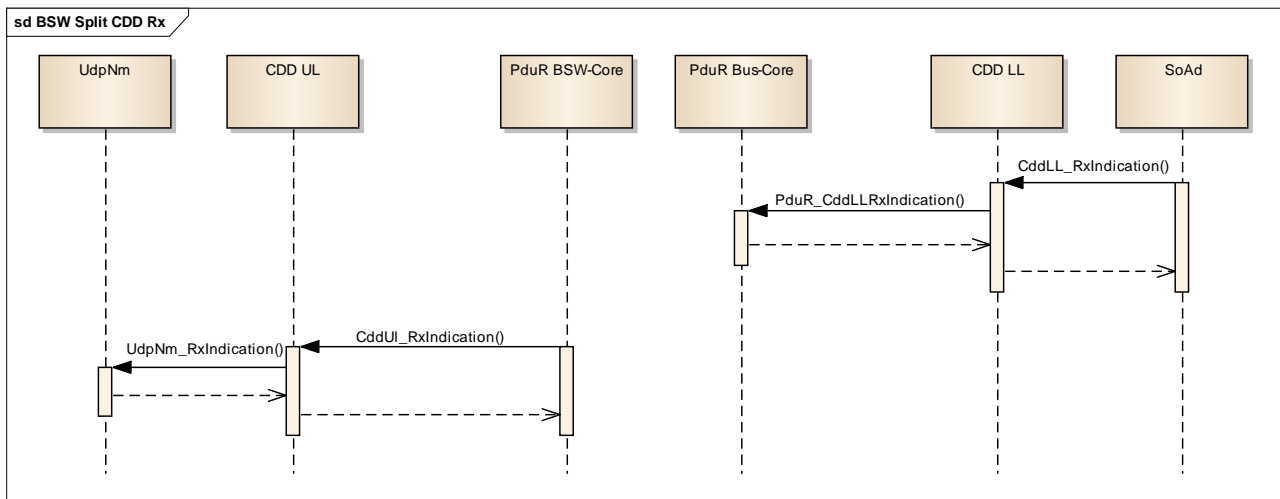


Figure 7-8 BSW-Split – Example CDD Reception

> Transmission

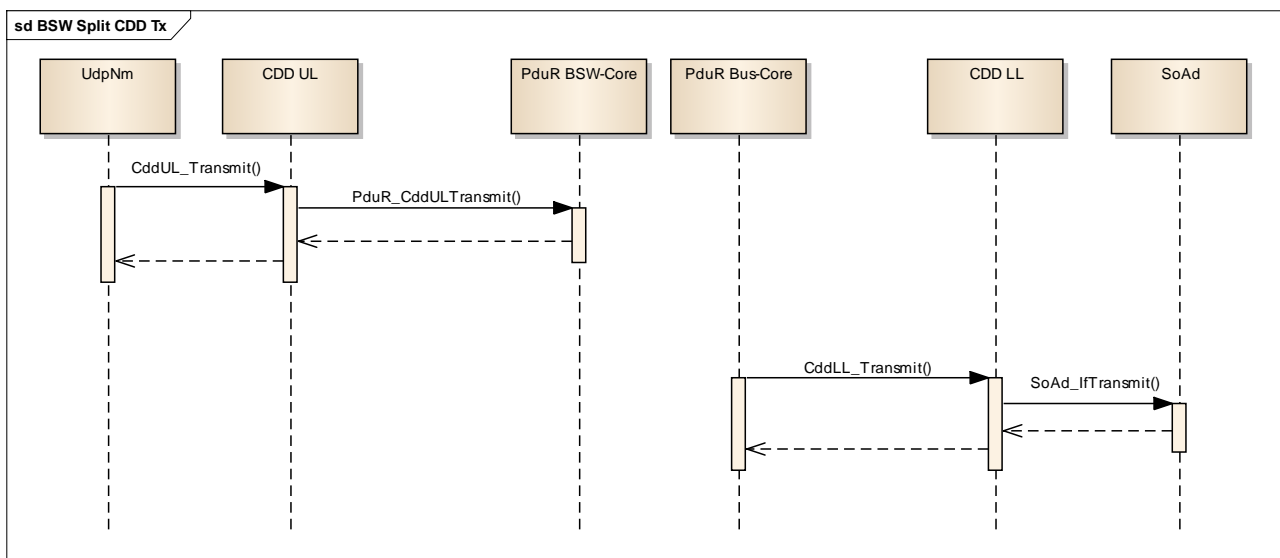


Figure 7-9 BSW-Split – Example CDD Transmission

> Transmission confirmation

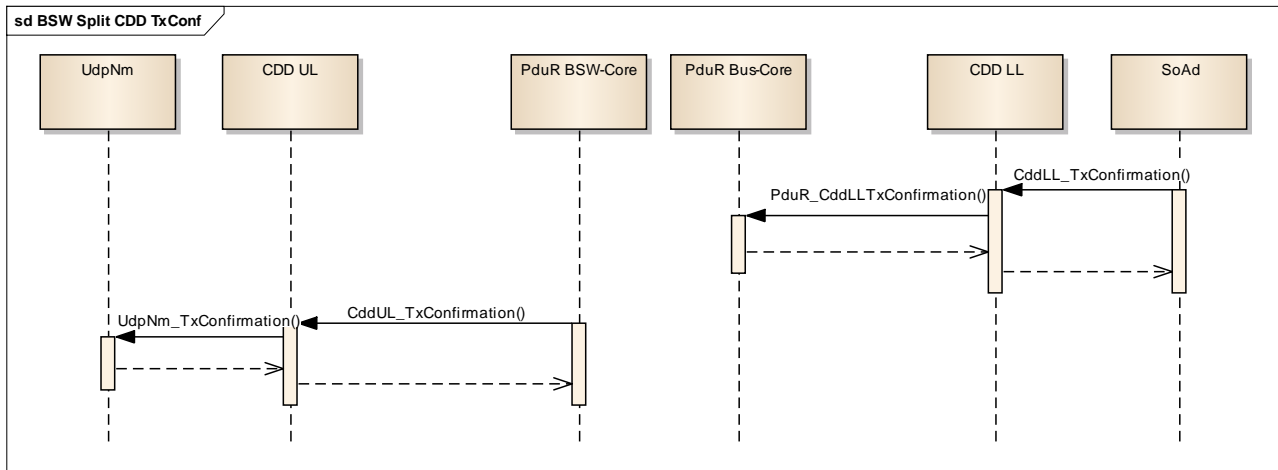


Figure 7-10 BSW-Split – Example CDD Tx Confirmation

7.3.6 Other Modules (BswM, Sd, NvM...)



Workaround

This chapter describes one workaround. It is the decision of the integrator to choose this workaround or devise an alternative.

The following modules invoke their APIs over core boundaries but are not multi-core capable yet:

- > ComM
- > BswM
- > EthSM
- > EthTrcv
- > Eth
- > EthIf
- > Spi
- > NvM
- > Sd

In this example, the affected API calls are handled as remote procedure calls: The call to a module on a different core is redirected to a wrapper (receiver) which notifies the associated wrapper (worker) on the target core. This wrapper takes the function arguments from the receiver and calls the target API.



Caution

Reference parameter might point to a memory on which the worker has no access (e.g. stack). In such cases either the MPU setting needs to be adjusted, or the parameter copied and buffered by the receiver wrapper.

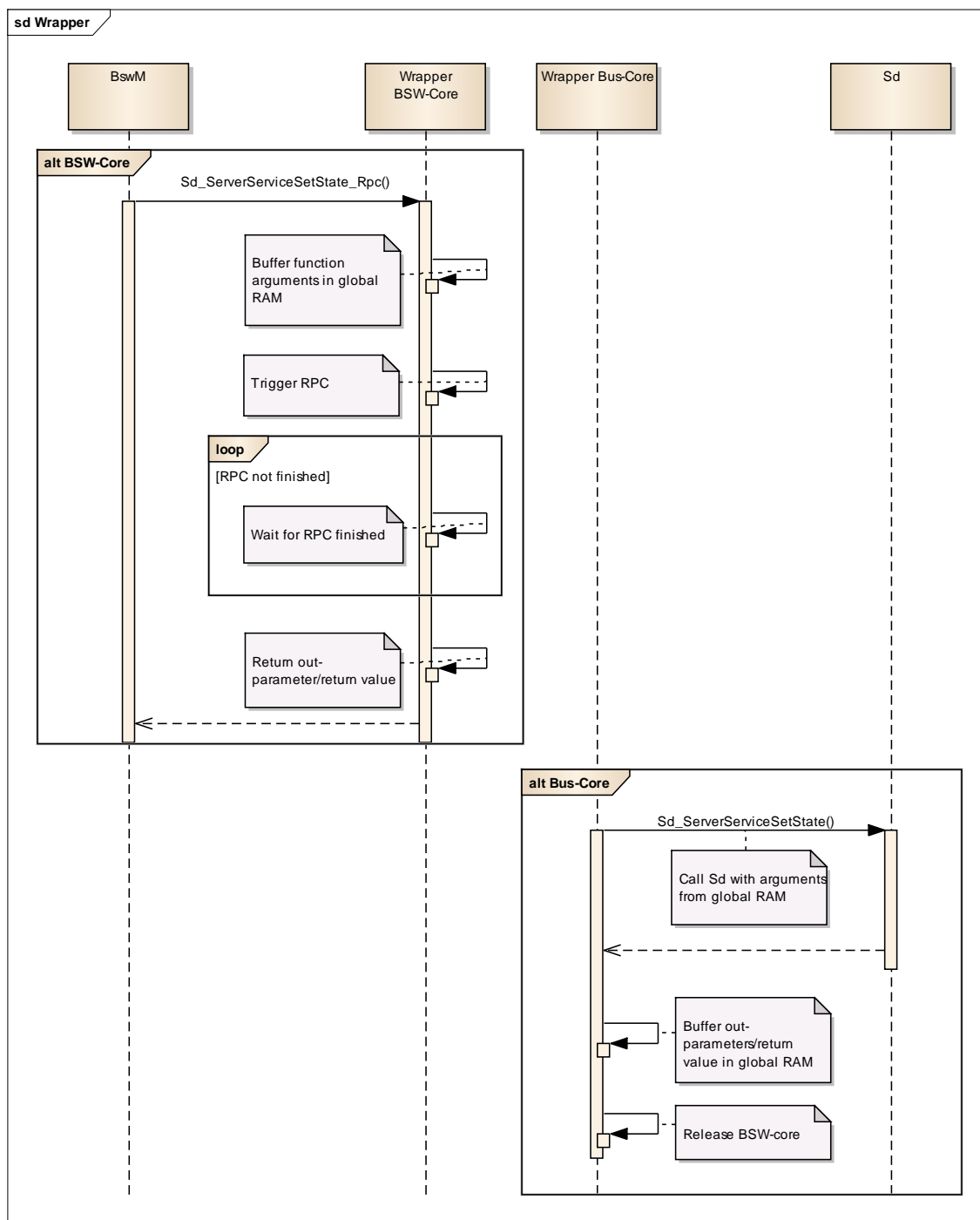


Figure 7-11 BSW-Split – Example RPC Wrapper

In this example, the target core is notified via an OS IOC. The worker is executed in the context of the IOC Receiver Pull Callback function. Other methods like OS Events, user implemented traps etc. are of course also valid options.

If there are no out parameter and return values, the caller core must not necessarily wait while the worker wrapper component is being executed. However, if it does not wait, the system timing behavior is different, because the remote function might still be waiting to be executed although the caller core has already finished its call.

7.3.7 Initialization

The BSW initialization is usually done by EcuM and the BswM. BswM however is not multi-core capable. As consequence, BswM actions which affect Ethernet related modules must be either decoupled (refer to chapter 7.3.5.2) or removed from the BswM config and realized by integration code.

Integration code for module initialization can be executed at three different stages:

1. main() context, right after the cores startup-code
2. EcuM init lists zero and one
3. Startup task context

Analog to the BswM module initialization list, which is executed after the OS start, the Ethernet related modules are initialized in the bus-core's init-task by integration code.

7.3.8 PduR Appl Spinlocks

The PduR uses spinlocks to protect its buffers and the TP, buffered and trigger transmit routings paths. However due to the circumstances (spinlocks are acquired nested and in the context of locked interrupts), OS spinlocks cannot be used. Hence a user implementation for the following APIs is required:

- > Std_ReturnType Appl_GetSpinlock(uint32 lockVar)
- > Std_ReturnType Appl_ReleaseSpinlock(uint32 lockVar)
- > uint32 Appl_GetSpinlockInitVal(void)

The PduR owns the locking variables which represent the lock. In the implementation, the following functionality must be realized atomically:

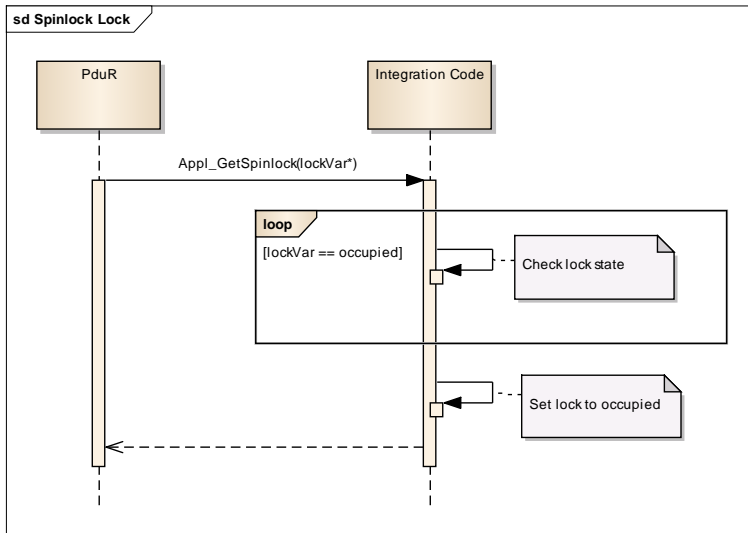


Figure 7-12 BSW-Split – Example PduR Appl_GetSpinlock

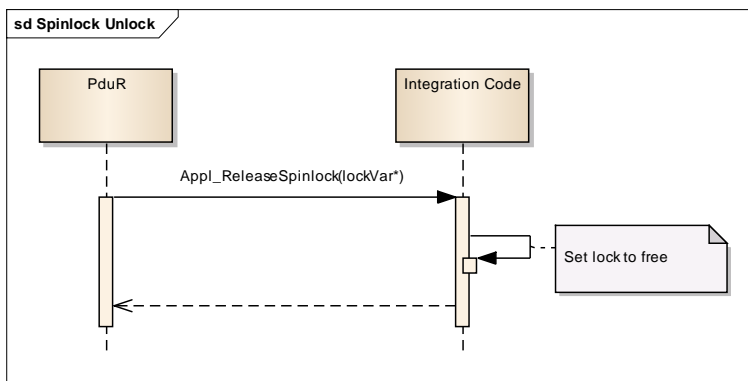


Figure 7-13 BSW-Split – Example PduR Appl_ReleaseSpinlock

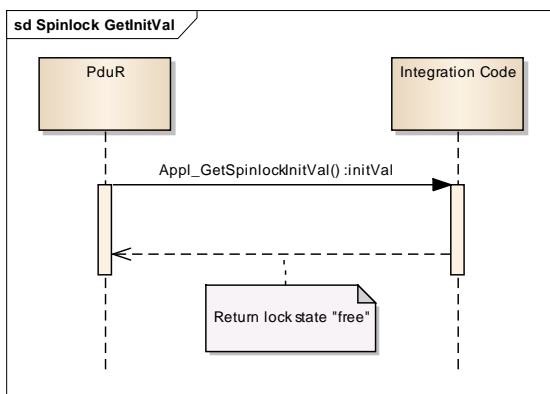


Figure 7-14 BSW-Split – Example PduR Appl_GetSpinlockInitVal

All three APIs must be implemented specifically for the given controller and compiler combination. Reason for the API Appl_GetSpinlockInitVal is that the lock states can be represented differently on different platforms. For example, **occupied** is represented as value **1** on one platform whereas it is **0** on another. To pass this information to the PduR, it

calls the function `Appl_GetSpinlockInitVal` for all locks during initialization, setting them to **free**.

**Caution**

The sequence showed for PduR `Appl_GetSpinlock` must be executed atomically. Otherwise, data consistency is not guaranteed.

7.3.9 Memory Mapping

Memory mapping follows the general description given in chapter 3. For performance reasons, the objects (code, data) of the Ethernet related modules should be mapped to core-local memories of the corresponding core if applicable.

In the simplest case, code is mapped so that all cores can execute code without restrictions. However, there are advanced μ C platforms with multiple flash banks/units. In these cases, the code must be mapped so that it is accessible by both cores

Additionally, the MPU might prevent execution. If this is the case, an adaptation of the MPU config is required, or a new flash section for the shared code must be created.

For the different modules, the requirements regarding memory mapping are:

- > PduR:
 - > Data must be globally read and writable for BSW core and the relocated bus-core
 - > Code section must be read and executable from both cores
- > RPC
 - > RAM of RPC components must be mapped to global RAM. Both cores require write access.
 - > RPC-receiver must reside in Flash, in which can be called by the caller BSW module
 - > RPC-worker must reside in Flash, from which it can call the callee BSW module
- > CDD
 - > LL CDD must reside in Flash, from which it can call the Ethernet modules
 - > UL CDD must reside in Flash, in which it can be called by the BSW-code

8 Application SWC Design Recommendations

This chapter gives a collection of basic architectural recommendations and general hints for the development of efficient multi-core application software. However, since software requirements differ vastly, there is no universal approach which can be applied to all projects. Furthermore, the behavior of the BSW and Rte depends on a huge variety of configuration parameters and application design aspects. Giving a complete description would easily exceed the scope of this document.

For this reason, this chapter is meant to be a pool of general guidelines from which the most applicable- and suitable items can be picked during development.

8.1 Code Design

This chapter contains the guidelines for designing the software functionality.

- > It is not possible to map Runnables of a SWC to different cores. Hence, SWC should be designed to bundle only those Runnables which are going to be mapped to the same core.
- > Bundle tightly coupled collections of SWC in Compositions. Hence re-locating to other cores in the scope of load balancing can be achieved simpler.
- > Favor cyclic IPO (Input-Process-Output) design patterns (e.g. Simulink models). IPO supports simple distribution of the different steps to different cores. Furthermore, cyclic execution and hence communication, can be realized with less overhead by the Rte than event driven communication.

8.2 Inter-Core Data Flow

Inter-core data flow is always expensive in terms of overhead runtime and should therefore be reduced to an absolute minimum. Depending on the type of data flow (e.g. between SWC or between SWC- and BSW) the communication path is more or less complex and hence runtime intensive.

The guidelines stated in the following chapter focus on keeping the overhead runtime to a minimum:

- > Minimize access from a remote core to COM signals to an absolute minimum. Crossing the core-boundary to COM is runtime intensive. Especially for write accesses.
- > Make use of simple, atomically read- and writable data types in Send/Receive (S/R) port interfaces instead of complex types. Port interfaces with native data types are realized as simple variables in shared memory with no additional protection whatsoever whereas complex data types require Spinlocks.
- > Deduction 1: Disaggregate complex data types of S/R port interfaces and use several native data type elements instead. This however is only sensible if data consistency is either less critical or still given by other mechanisms such as scheduling sequences
- > Deduction 2: If complex data types are crucial for securing data consistency, aggregate data elements of several S/R port interfaces which can be read- or written

simultaneously. Thus, only a single large data element has to be transferred via IOC instead of several smaller elements. This results in a better overhead to payload ratio in the IOC communication.

- > Prefer un-queued (Last-Is-Best) communication in S/R Port Interfaces (Read Data) instead of queued communication (Receive Data). The queue-mechanism needs to be protected from concurrent access which is done by Spinlocks.
- > Use Buffered Port access for S/R Port Interfaces (Implicit Port access). Hence Port data is buffered in a runnable-exclusive, core-local temporary memory. For Receive Port Interfaces, the data is fetched and buffered immediately before the runnable is called while data transmitted via Send Port Interfaces is written back not until the runnable has finished. As result the runtime-costly crossing of core-boundaries is needed only once for each Port Interface. No matter how often the Port Interface is actually accessed by the Runnable's application code. Drawback is of course that more RAM is needed.
- > Schedule Runnables which cyclically access S/R Port Interfaces with cycle times as long as tolerable. The result is less IOC-usage per given period of time and ultimately less overhead runtime
- > Avoid Runnable activations triggered by data related events of S/R Port Interfaces like:
 - > On Data Reception
 - > On Data Reception Error
 - > On Data Send Completion

These require OS mechanisms (events, task activations) on remote cores to schedule Runnables which adds runtime overhead but it may reduce spinning time.

8.3 Software Distribution/Mapping

This chapter describes approaches for mapping respectively distributing the Runnables/SWC to cores.

- > Map all SWCs of a single ECU-function to tasks on the same core. This follows the assumption that coupling between ECU-functions is rather low and requires fewer Port Interfaces across core boundaries than an approach in which the SWC of a function are distributed.
- > Map SWC which make heavy use of BSW-services (e.g. communication) to tasks bound to the BSW-core. This eliminates the need to cross core boundaries via RPC for each call to the BSW. Especially for frequently used interfaces, even small overheads can sum up considerably.
- > Map computational intensive algorithms/SWC to tasks to a remote-core. This is especially useful if CPU-load of the BSW-core shall be reduced. This has a positive effect as soon as the runtime of the relocated SWC is greater than the overhead of the core-crossing of its Port Interfaces.
- > Map software parts with chained computation order to the same core. This especially applies to configurations in which the order is defined by the mapping of Runnables to

tasks. Simple sequences on the same core- and task with the same cyclic timing can be realized very efficiently as a sequence of function calls. Distributed over cores however, OS-mechanisms are required to ensure the proper execution order.

- > Map entire Simulink models to remote cores. Such models with their comparably long runtime are ideally suited for multi-core especially when all input data is read once at the beginning of the computation step and written back once after its completion.

8.4 Configuration Aspects

Avoid non-preemptive tasks. Especially when (time critical) execution-sequences are distributed over several cores. Otherwise, due to the OS's scheduling policy, a high priority-task may have to wait for a low-priority, non-preemptive task to finish on a remote core. On the other hand, a non-preemptive task may require more effort to handle data consistency issues.

9 Glossary and Abbreviations

9.1 Glossary

Term	Description
EAD	Embedded Architecture Designer; generation tool for MICROSAR components
GENy	Generation tool for CANbedded and MICROSAR components

Table 9-1 Glossary

9.2 Abbreviations

Abbreviation	Description
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BSW	Basis Software
DEM	Diagnostic Event Manager
DET	Development Error Tracer
EAD	Embedded Architecture Designer
ECU	Electronic Control Unit
HIS	Hersteller Initiative Software
ISR	Interrupt Service Routine
MICROSAR	Microcontroller Open System Architecture (the Vector AUTOSAR solution)
RPC	Remote Procedure Call
PSPORT	Provide Port
RPORT	Require Port
Rte	Runtime Environment
SRS	Software Requirement Specification
SWC	Software Component
SWS	Software Specification

Table 9-2 Abbreviations

10 Contact

Visit our website for more information on

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

www.vector.com