Document Title	Guide to Modemanagement
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	440
Document Classification	Auxiliary

Document Version	1.0.0.
Document Status	Final
Part of Release	4.0
Revision	3

Document Change History			
Date	**Version**	**Changed by**	**Description**
27.10.2011	1.0.0	AUTOSAR Administration	Initial release

Advice for users

AUTOSAR specifications may contain exemplary items (exemplary reference models, "use cases", and/or references to exemplary technical solutions, devices, processes or software).

Any such exemplary items are contained in the specifications for illustration purposes only, and they themselves are not part of the AUTOSAR Standard. Neither their presence in such specifications, nor any later documentation of AUTOSAR conformance of products actually implementing such exemplary items, imply that intellectual property rights covering such exemplary items are licensed under the same rules as applicable to the AUTOSAR Standard.

Table of Contents

References

[1] Specification of ECU State Manager with fixed state machine
AUTOSAR_SWS_ECUStateManagerFixed.pdf

[2] Software Component Template
AUTOSAR_TPS_SoftwareComponentTemplate.pdf

[3] Meta Model
AUTOSAR_MMOD_MetaModel.eap

[4] Basic Software Module Description Template
AUTOSAR_TPS_BSWModuleDescriptionTemplate.pdf

[5] Specification of Basic Software Mode Manager
AUTOSAR_SWS_BSWModeManager.pdf

[6] Glossary
AUTOSAR_TR_Glossary.pdf

1 Introduction

This document is a general introduction to AUTOSAR Mode Management for the Release 4.0.3 onwards. Its main purpose is to give users as well as developers of AUTOSAR an detailed overview of the different aspects of AUTOSAR mode management.

Chapter 2 explains the basic mode management concepts e.g. modes in general, how mode switches are implemented, roles of mode managers and mode users etc. It secondly gives an introduction to Application Mode management and the dependencies to Basic Software Mode management, which are closely related.

The `Basic Software Modemanager` is the central mode management module in AUTOSAR R4.0. It is configurable to a high degree. How this configuration can be achieved is the topic of chapter 3.

Chapter 4 than deals with migration strategies from fixed ECU Management as it was used in AUTOSAR R3.1 [1] to the new approach of ECU management of AUTOSAR 4.0

1.1 Further Work

Due to complexity and broad scope of this topic there are still some uses cases which are not yet described here in full detail. These issues will be enhanced in further releases.

- ECUs as Gateways
- Communication management for Flex Ray
- Communication management for Ethernet
- Communication management for Lin (including schedule table switching)
- DCM Routing path groups
- BSWM configuration for multicore ECUs
- DCM Session Control has to be added

[1] and in R4.0 with the ECU Statemanager with fixed state machine[1]

2 Overall mechanisms and concepts

This chapter gives an overview of the concept of modes and a short definition of states in AUTOSAR. Defintions of the terms mode and state can be found in chapter 5.1 A mode can be seen as the current state of an ECU[1] wide, global variable, which is maintained by the RTE respectively the Schedule Manager. The possible assignments of a mode are defined in `ModeDeclarationGroups`, which are defined in the AUTOSAR Software Component Template [2]. Modes can be used for different purposes. First of all modes are used to synchronize Software Components and Basic Software Modules. Via modes specified triggers can be enabled and disabled, and consequently the activation of `ExecutableEntities` can be prevented. Also `ExecutableEntities` can be triggered explicitely during a Mode Switch. On the other hand mode switches can explicitly trigger executable entities during transition from one mode to another. For example the RTE can activate an `OnEntry ExecutableEntity` to initialize a certain resource before entering a specific mode. In this mode the triggers of this `ExecutableEntity` are activated. If the mode is left the `OnExit ExecutableEntity` is called, which could execute some cleanup code and the triggers would be deactivated.

2.1 Declaration of modes

The Software Component Template defines a generic mechanism for describing modes in AUTOSAR. Modes are defined via `ModeDeclarations`. A `ModeDeclaration` represents a possible assignment of the current state of a global variable. E.g in ECU state management there may exist the ModeDeclarations STARTUP, RUN, POST_RUN, SLEEP.

A `ModeDeclarationGroup` groups several `ModeDeclaration`s in a similar way as an enumeration groups literals. In the given example this could be the `ModeDeclarationGroup` ECUMODE. For each `ModeDeclarationGroup` an `InitialMode` has to be defined, which is assigned to the variable at startup. Figure 2.1 shows an excerpt of the AUTOSAR Metamodel [3] with the relationships of `ModeDeclaration`s, `ModeDeclarationGroup`s and `Executable Entitie`s.

[1] In R4.0 this is limited to a single partition

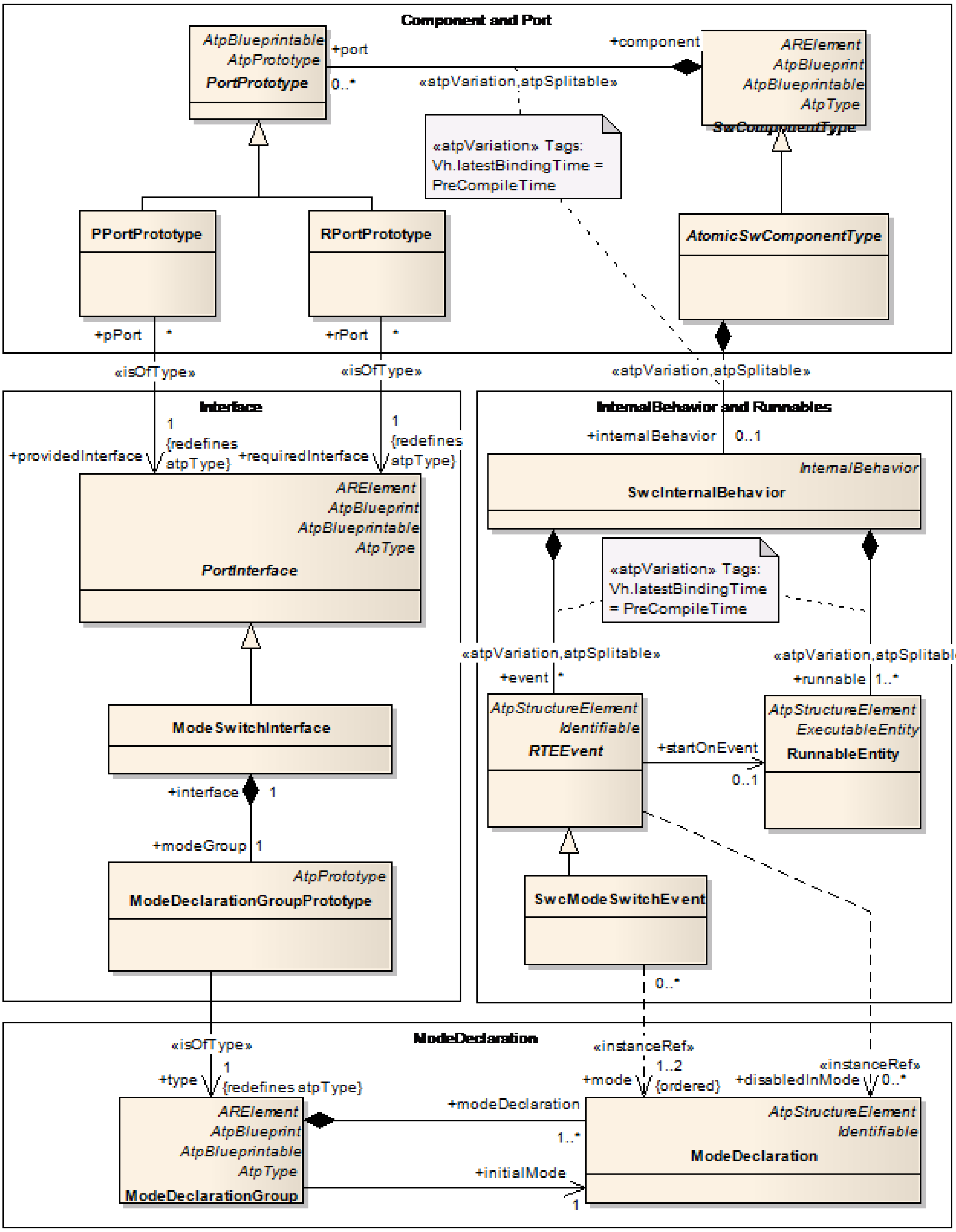

Figure 2.1: Excerpt of Metamodel regarding Modes

2.2 Mode managers and Mode users

In mode management there are two parties involved: *Mode managers* and *mode users*. Responsible for switching modes are *Mode managers*, which are the only instances able to change the value of the global variable. A mode manager is either a Software Component, which provides a `ModeREQUESTPort` or a *Basic Software Module*, which either provides also a `ModeREQUESTPort` in its `Software Component Description` or a `ModeDeclarationGroup` in its `Basic Software Module Description`. `Mode users` are informed of `Mode switches` via well-defined mechanisms and have the possibility to read the currently active mode at any time. If a Mode user wants to change into a different mode it can request a `Mode switch` from the corresponding `Mode manager`.

2.3 Modes in the RTE

The AUTOSAR Runtime Environment implements the concept of modes. For this purposes it creates for each `ModeDeclarationGroupPrototype` of an `Atomic Software Component` a so called `ModeMachineInstance`. A `ModeMachineInstance` is a state machine whose states are defined by the `ModeDeclarations` of the respective `ModeDeclarationGroup`.

Figure 2.2 depicts the interaction of `ModeDeclarationGroupPrototypes` Mode managers and Mode users. Note that the mode switch ports of the mode users are not directly connected to the corresponding `PPorts` of the mode managers but instead are connected to the mode machine instances of the RTE. This is important to understand the mechanism of mode switching inside the RTE.

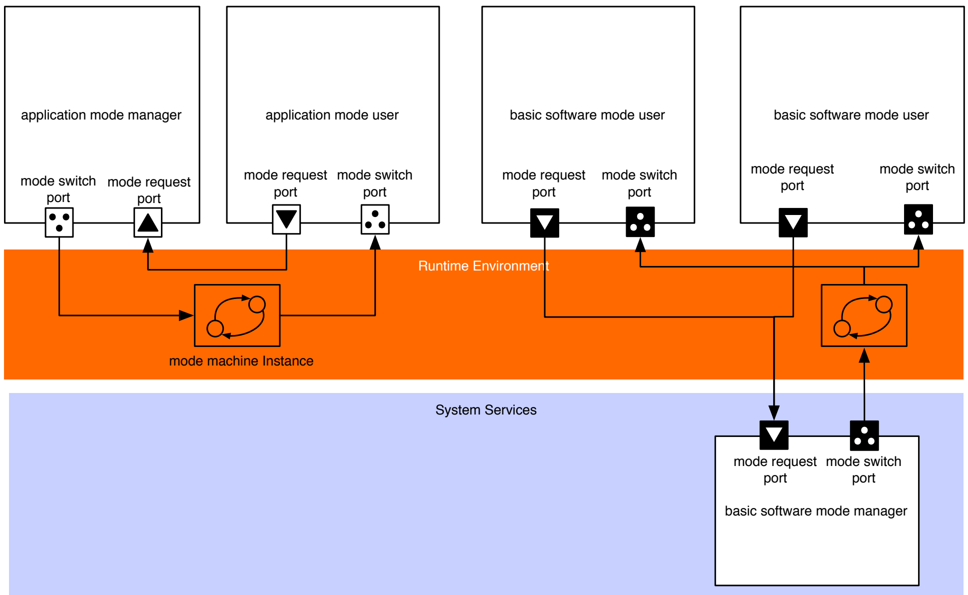

Figure 2.2: The RTE instantiates for each ModeDeclarationGroupPrototype a ModemachineInstance

Previous versions of the Basic Software Modules especially the ECU state manager module have differentiated between ECU states and ECU modes. ECU modes were longer lasting operational ECU states that were visible to applications i.e. starting up, shutting down, going to sleep and waking up. The ECU Manager states were generally continuous sequences of ECU Manager module operations terminated by waiting until external conditions were fulfilled. Startup1, for example, contained all BSW initialization before the OS was started and terminated when the OS returned control to the ECU Manager module. With flexible ECU management the ECU state machine is implemented as general modes under the control of the BSW Mode Manager module. To overcame this terminology problem states are used only internally and are not visible to the application. For interaction with the application the basic software has to use modes.

2.4 Modes in the Basic Software Scheduler

The `Basic Software Scheduler` provides for `Basic Software Modules` a similar mechanism for mode communication as the RTE provides it for Software Components. If a `Basic Software Module` provides a `ModeDeclarationGroupPrototype` as `providedModeGroup` in its Basic Software Module Description the `Basic Software Scheduler` instatiates a `ModeMachineInstance`. Consequently for this `Basic Software Module` a `SchM_Switch` API is provided, which enables this module to initiate a `Mode switch`. `Mode users` have to reference the `ModeDeclarationGroupPrototype` as `requiredModeGroup` and will get a `SchM_Mode` API to read the mode, which is currently active. `Mode requests` between `Basic Software Modules` can be comunicated directly via function calls, as `Basic Software Modules`.

Another possibility for a `Basic Software Module` acting as a `Mode user` to get informed about mode switches, is to register a `BSW Module Entry`, which is triggered by a `Mode Switch Event` (see also [4]).

2.5 Communication of modes

The Software Component Template differs the following distinctive types of mode communication between Mode managers and Mode users.

- Mode Switch: A Mode Switch is the communication of a current mode transition from one mode to another. Mode Switches are always initiated by Mode Managers.
- Mode Request: A Mode Request is the request of a mode user to the Mode Manager to enter a certain mode. Note that it is not guaranteed that the Mode Manager will enter this mode. Moreover he has to arbitrate all requests from the Mode Users and decide which mode he will enter.

Furthermore, the concept of Mode Proxies and information about communication of modes on multi core ECUs is given.

2.5.1 Mode switch

As every other communication between Software Components or between Software Components and Basic Software Modules, Modes are communicated via PortPrototypes. Each PortPrototype has to be typed by a PortInterface. In case of mode communication there exist so called ModeSwitchInterfaces, which are PortInterfaces. These are shown in Figure 2.3. Each ModeSwitchInterface has exactly one ModeDeclarationGroupPrototype which consists of multiple ModeDeclarations. Any ModeDeclaration represents one mode of the ModeDeclarationGroup. One of these is defined as the initial mode.

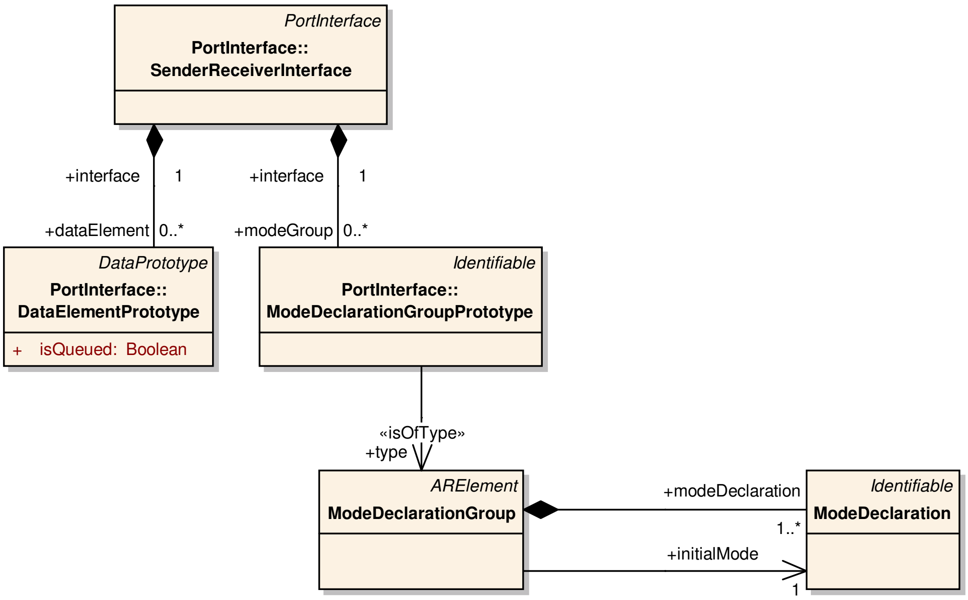

Figure 2.3: ModeSwitchInterface

These Mode switches are necessary because Software Components need to be capable of reacting to state changes initiated by a ModeManager. Depending on the configuration there are two mechanisms available how a Software Component can react on a mode change.

1. A ModeSwitchEvent can trigger a OnExtry, OnTransition or OnEntry-Runnable.
2. An RTEEvent can be disabled in a certain mode and consequently prevent the execution of accordant ExecutableEntities.

2.5.2 Mode request

Mode requests are distributed on the way from the Mode Requester (Mode Arbitration SWC or a generic SWC) to the ModeManager. The mode managers on each ECU then have to decide and initiate the local mode switch. Thus the arbitration result is communicated only locally on each ECU using RTE mode `switch mechanism`.

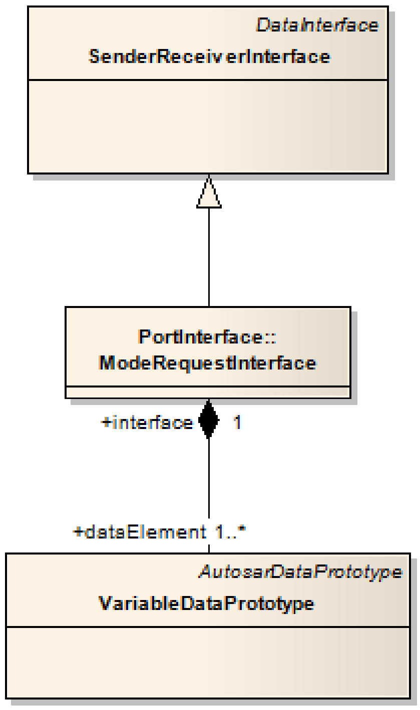

Figure 2.4: ModeRequestInterface

For `Mode requests`, the communication of modes works slightly differently as for `Mode switches`: Without `ModeDeclarationGroup`s. This is illustrated in Figure 2.4.

The request of modes is done via `ModeRequestInterface`s which are standard Autosar `SenderReceiverInterface`s with that special type. Contrarily to `ModeSwitchInterface`s the requested mode is not given by a `ModeDeclarationGroup` but by a `VariableDataPrototype` that has to contain an enumeration. This enumeration consists of a set which contains the modes that can be requested.

Mode requests can be distributed in the whole system. For Application and Vehicle Modes, the requests of the Mode Requester have to be distributed to all affected ECUs. This implies a 1:n-connection between the Mode Requester and the Mode Managers. In AUTOSAR this is only possible with Sender-Receiver Communication. The mode manager only requires the information about the requested mode and not the mode switch from the mode requester. The Mode Manager has one Sender-Receiver port for each mode requester. To actually transmit the signal, COM shall use a periodic signal with signal timeout notification to RTE. The Mode Manager will use the data element outdated event to release a Mode Request. An action shall only be carried out if it brings the effected interface into a different state. E.g. having to 'start' actions, only the first one shall be effective. The second one needs to be filtered out.

2.5.3 Conformance of mode switches and mode requests

As stated above, the ModeSwitchInterfaces work with ModeDeclarationGroups whereas ModeRequestInterfaces takes parameter via VariableDataPrototypes containing enumerations.

The configuration utility is in duty to ensure with respect to consistency the equivalence of represented data in both representations. That means that the elements of the enumeration must precisely match the elements of the ModeDeclarationGroup. Or formulated another way: All modes available in one of the interfaces must also be available in the other one.

2.5.4 Mode proxies

Currently AUTOSAR has a constraint that only local SoftwareComponents are allowed to communicate with ServiceComponents. So it is not possible that a SoftwareComponent can request modes from a remote e.g Basic Software Mode Manager. To overcome this limitation so called ServiceProxyComponentType were introduced in AUTOSAR Release 4.0. Figure 2.5 depicts this concept.

For the application software and the RTE a ServiceProxySoftwareComponentType behaves like a "normal" AtomicSwComponentType, but it is actually a proxy for an AUTOSAR Service. This means that on the one side it has to communicate over service ports with the ECU-local ServiceSwComponentType it represents. On the other side it has to offer the corresponding PortPrototypes to the ApplicationSwComponentTypes. In the meta-model, the ServiceProxySwComponentType does not differ from an ApplicationSwComponentType except by its class. It is up to the implementer to meet the restrictions imposed by the semantics as a proxy. The main difference between a ServiceProxySwComponentType and an ApplicationSwComponentType is on system level: A prototype of a ServiceProxySwComponentType can be mapped to several ECUs even if it appears only once in the VFB system, because such a prototype is required on each ECU, where it has to address a local ServiceSwComponentType. As a result of this, a ServiceProxySwComponentType can only receive but not send signals over the network. (see also [2]).

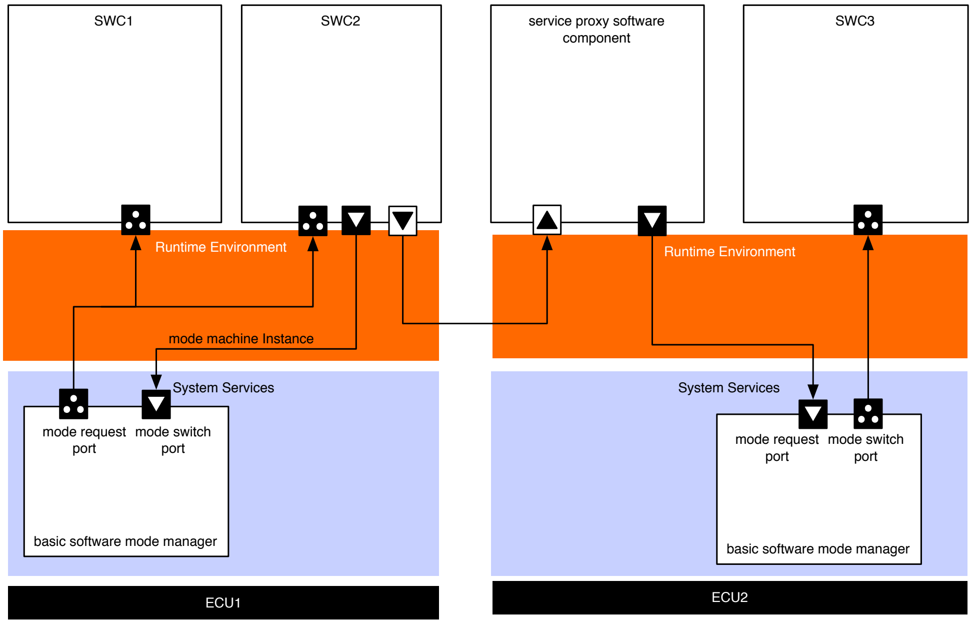

Figure 2.5: Communication via ServiceProxySwComponents

2.5.5 Mode communication on multi core ECUs

The RTE does not synchronize ModeMachineInstances over the different partitions of an ECU. rte_sws_2724 states that the RTE shall reject configurations where one `ModeDeclarationGroupPrototype` of a provide port is connected to `ModeDeclarationGroupPrototypes` of require ports from more than one partition. Consequently all `ModeUsers` of a `ModeDeclarationGroupPrototype` have to live inside a single partition. Note that the `ModeManager` of the `ModeDeclarationGroupPrototype` can of course exist in another partition as shown in Fig. 2.7

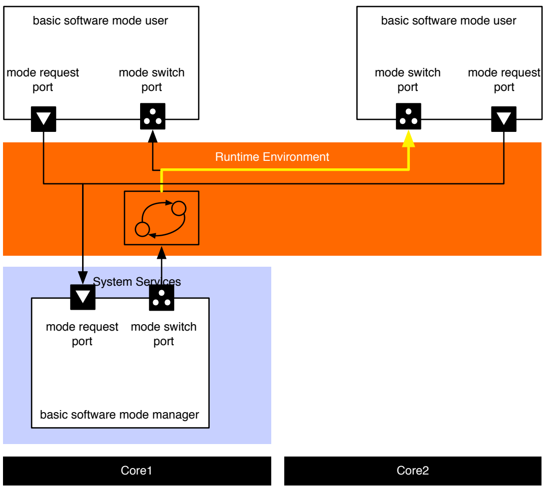

Figure 2.6: Invalid configuration

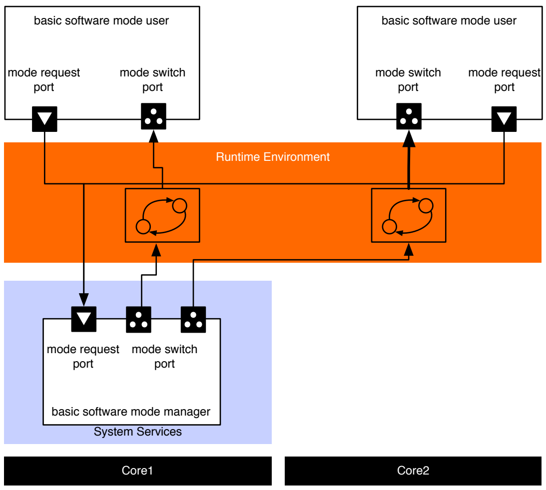

Figure 2.7: Corrected version according to [rte_sws_2724]

This limitation has a deep impact on mode managers with mode users on different cores. The mode manager has to provide a dedicated `ModePort` for each partition in which one or more of it's mode users are located. To trigger a mode change it has to call Rte_switch for each mode port separately. If configured it will also get an separate Mode_Switch_Acknowldegement from each `ModeMachineInstance`. This means that the possible mapping of mode users and mode managers to different core has to be taken into account to some extend during design time of the Software Components.

3 Configuration of the Basic Software Modemanagers

The BSW Mode Manager is the module that implements the part of the Vehicle Mode Management and Application Mode Management concept that resides in the BSW. Its responsibility is to arbitrate mode requests from application layer Software Components or other Basic Software Modules based on rules, and perform actions based on the arbitration result.

From an functional point view the BSWM is responsible to put the Basic Software in a state so that the Basic Software can run properly and meet the functional requirements.

The configuration of the BSWM is very project- and ECU specific. Therefore it can not be standardized by AUTOSAR. Nevertheless it is expected that a BSWM implementation behaves in specific situations in a certain way . This chapter starts with an introduction on the general concept of the BSWM, which is more or less a execution environment for rules described by the user. Afterwards typical scenarios in the lifecycle of an ECU are described and examples are given how the BSWM could be configured.

3.1 Process how to configure and integrate a BSWM

The configuration and integration of a BSWM into an ECU project consists of the same steps as for other Basic Software Modules. Nevertheless it is described for a better understanding of the next steps. In general the following steps have to be taken:

1. Create a ECUC configuration of the module. the configuration contains:
   (a) the necessary `ModeRequestSources`
   (b) the provided `ModeSwitchPorts`
   (c) a description of the `Rules` and `ActionLists`
2. The configuration is used as input for the module generator, which creates
   (a) a `SoftwareComponentDescription` of the AUTOSAR Interface
   (b) the implementation of the module[1]
3. The last step is to integrate the Module into the ECU by connecting the ports of the `Software Components` with the corresponding ports of the BSWM.

3.2 Semantics of BSWM Configuration: Interfaces and behavioral aspects

In general the BSWM can be seen as a state machine, which is defined by its interface and a behavioral description. The input actions of this state machine are mode

[1] This documents assumes that the Implementation of the BSWM is generated to a large extend.

requests. In real implementations these mode requests can be of different types (C-API calls, mode requests via RTE, mode notifications via RTE, etc.) but are treatened internally in the same way. If a mode is requested the internal mirror of this BswMModeRequestSource is updated and depending on the configuration a rule evalutation is triggered, which results in the exuction of predefined action lists.

BswMActionListItems can be of similar kinds as mode requests: simple API calls and mode switches via RTE or the Schedule Manager.

3.2.1 Interface of the BSWM

The interface is defined by the BswMModeRequestSource and the BswMActionListItem containers.

BswMModeRequestSource is a ChoiceContainer, which can be of the following kinds:

1. C-APIs, which are defined in the specification of the BSWM. BasicSoftwareModules can directly call C-APIs from the BSWM, which will translate it internally into a ModeRequest. For example a call to the API

   ```
   BswM_CanSM_CurrentState(
           NetworkHandleType Network,
           CanSM_BswMCurrentStateType CurrentState
   )
   ```

   has to be mapped to different ModeRequestPorts depending on the parameter Network, which identifies the channel on which the event occurred. The parameter CurrentState then contains the mode which is requested. The follwing mode request types are defined:

   (a) BswMCanSMIndication
   (b) BswMEcuMWakeupSource
   (c) BswMEthSMIndication
   (d) BswMFrSMIndication
   (e) BswMLinSMIndication
   (f) BswMLinScheduleIndication
   (g) BswMLinTpModeRequest
   (h) BswMNvMRequest
   (i) BswMWdgMRequestPartitionReset
   (j) BswMComMIndication
   (k) BswMGenericRequest

2. RPorts typed by a SenderReceiverInterface.
   (a) BswMSwcModeRequest: For each container of this type the BSWM has to create a corresponding RPort in its Service Component Description.
3. RPorts typed by a ModeSwitchInterface.
   (a) BswMSwcModeNotification: For each container of this type the BSWM has to create a corresponding RPort in its Service Component Description. As it is typed by a ModeSwitchInterface the BSWM acts as a mode user of this ModeMachineInstance and is informed if the mode manager performs an rte_switch.
4. RequiredModeDeclarationGroupPrototypes
   (a) BswMBswModeNotification: For each container of this type the BSWM has to create a corresponding RequiredModeDeclarationGroupPrototype in the role RequiredModeDeclarationGroup in its Basic Software Module Description. The BSWM also acts as a mode user, but the ModeMachineInstance is maintained by the Schedule Manager. The BSWM therefore gets informed if the mode manager e.g. another Basic Software Module performs a SchM_Switch call.

BswMActionListItems can be of the following kinds:

1. C-APIs from other BSWM Modules, which are called directly during the execution of an ActionList.
   (a) BswMUserCall
   (b) BswMComMAllowCom
   (c) BswMComMModeSwitch
   (d) BswMDeadlineMonitoringControl
   (e) BswMLinScheduleSwitch
   (f) BswMNMControl
   (g) BswMPduGroupSwitch
   (h) BswMPduRouterControl
   (i) BswMResetSignalInitValues
   (j) BswMRteSwitch
   (k) BswMSchMSwitch
   (l) BswMTriggerIPduSend
   (m) BswMTriggerSlaveRTEStop
   (n) BswMTriggerStartUpPhase2

2. `PPort`s typed by a `ModeSwitchInterface`
   (a) BswMRteSwitch : For each container of this type the BSWM has to create a corresponding `PPort` in its Service Component Description.
3. `ProvidedModeDeclarationGroupPrototypes`
   (a) BswMSchMSwitch: For each container of this type the BSWM has to create a corresponding `ProvidedModeDeclarationGroupPrototype` in the role `ProvidedModeDeclarationGroup` in its Basic Software Module Description. The BSWM also acts as a mode manager, but the `ModeMachineInstance` is maintained by the Schedule Manager.

Listing 3.1: Configuration of a ModeRequestSource in pseudo code

```
request CanSMIndication Can1_indication {
  processing IMMEDIATE
  initialValue "CANSM_BSWM_NO_COMMUNICATION"
  source CanSM.CanStateManagerConfiguration.Can1StateManagerNetwork
}
```

3.2.2 Definitions of ModeDeclarationGroups

An example of the BswM configuration of `ModeSwitchInterface`s is shown in Listing 3.2. There is a `ModeDeclarationGroup` and a `ModeSwitchInterface` created. The `ModeSwitchInterface` uses the defined `ModeDeclarationGroup` as prototype where *exampleModes* is the short name of the `ModeSwitchInterface`.

Listing 3.2: Declaration of a ModeSwitchInterface

```
modeGroup exampleModeDeclarationGroup {
  Mode1,
  Mode2,
  Mode3
}

interface modeSwitch exampleModeSwitchInterface {
   mode exampleModeDeclarationGroup exampleModes
}
```

A configuration of a `ModeRequestInterfaces` that corresponds to the `ModeRequestInterfaces` of Listing 3.2 is shown as example in Listing 3.3. Out of this BswM configuration an *Arxml* description will be created which includes the mode declarations and interfaces. An excerpt of that arxml is shown in 3.4.

Listing 3.3: Declaration of a ModeRequestInterface

```
enum exampleModeEnumeration {
  Mode1,
  Mode2,
  Mode3
}
```

```
interface senderReceiver exampleModeRequestPort {
  data exampleModeEnumeration exampleModeRequest
}
```

Listing 3.4: Excerpt of the ModeRequestInterface's Arxml description

```
<SENDER-RECEIVER-INTERFACE>
  <SHORT-NAME>exampleModeRequestPort</SHORT-NAME>
  <IS-SERVICE>false</IS-SERVICE>
  <DATA-ELEMENTS>
  <VARIABLE-DATA-PROTOTYPE>
    <SHORT-NAME>exampleModeRequest</SHORT-NAME>
    ...
    <TYPE-TREF DEST="APPLICATION-PRIMITIVE-DATA-TYPE">
       exampleModeEnumeration</TYPE-TREF>
  </VARIABLE-DATA-PROTOTYPE>
  </DATA-ELEMENTS>
</SENDER-RECEIVER-INTERFACE>

...

<APPLICATION-PRIMITIVE-DATA-TYPE>
  <SHORT-NAME>exampleModeEnumeration</SHORT-NAME>
  <CATEGORY>VALUE</CATEGORY>
  <SW-DATA-DEF-PROPS>
    <SW-DATA-DEF-PROPS-VARIANTS>
      <SW-DATA-DEF-PROPS-CONDITIONAL>
        <COMPU-METHOD-REF DEST="COMPU-METHOD">exampleModeEnumeration_def</
           COMPU-METHOD-REF>
      </SW-DATA-DEF-PROPS-CONDITIONAL>
    </SW-DATA-DEF-PROPS-VARIANTS>
  </SW-DATA-DEF-PROPS>
</APPLICATION-PRIMITIVE-DATA-TYPE>

...

<COMPU-METHOD>
  <SHORT-NAME>exampleModeEnumeration_def</SHORT-NAME>
  <CATEGORY>TEXTTABLE</CATEGORY>
  <COMPU-INTERNAL-TO-PHYS>
  <COMPU-SCALES>
    <COMPU-SCALE>
    <LOWER-LIMIT INTERVAL-TYPE="CLOSED">0</LOWER-LIMIT>
    <UPPER-LIMIT INTERVAL-TYPE="CLOSED">0</UPPER-LIMIT>
    <COMPU-CONST>
      <VT>Mode1</VT>
    </COMPU-CONST>
    </COMPU-SCALE>
    <COMPU-SCALE>
    <LOWER-LIMIT INTERVAL-TYPE="CLOSED">1</LOWER-LIMIT>
    <UPPER-LIMIT INTERVAL-TYPE="CLOSED">1</UPPER-LIMIT>
    <COMPU-CONST>
      <VT>Mode2</VT>
    </COMPU-CONST>
    </COMPU-SCALE>
    <COMPU-SCALE>
```

```
    <LOWER-LIMIT INTERVAL-TYPE="CLOSED">2</LOWER-LIMIT>
    <UPPER-LIMIT INTERVAL-TYPE="CLOSED">2</UPPER-LIMIT>
    <COMPU-CONST>
      <VT>Mode3</VT>
    </COMPU-CONST>
    </COMPU-SCALE>
  </COMPU-SCALES>
  </COMPU-INTERNAL-TO-PHYS>
</COMPU-METHOD>
```

Every mode request to the BSWM has to be mapped to an restricted set of values, which allows the integrator the define the arbitration rules.

3.2.2.1 ModeDeclarationGroups defined by the standardized interface of the BSWM

The following `ModeDeclarationGroup`s are defined in the particular SWS documents of the Autosar specification as C-Enums. Nevertheless they are shown here in form of BswM configurations to enable a clear overview of the defined modes and act as a base for the rest of this document.

From the BswM's point of view there is no difference whether the modes are specified by the SWSs as C-Enums or as `ModeDeclarationGroup`s by BswM configuration.

Listing 3.5: Modes reported by the API BswM_ComM_CurrentMode

```
modeGroup ComM_ModeType{

  COMM_NO_COM_NO_PENDING_REQUEST,
  COMM_NO_COM_REQUEST_PENDING,
  COMM_FULL_COM_NETWORK_REQUESTED,
  COMM_FULL_COM_READY_SLEEP

}
```

Listing 3.6: Modes reported by the API BswM_ComM_CurrentPNCMode

```
modeGroup ComM_PncModeType{
  PNC_REQUESTED,
  PNC_READY_SLEEP,
  PNC_PREPARE_SLEEP,
  PNC_NO_COMMUNICATION,
  PNC_FULL_COMMUNICATION

}
```

Listing 3.7: Modes reported by the API BswM_CanSM_CurrentState

```
modeGroup CanSM_BswMCurrentStateType{
  CANSM_BSWM_NO_COMMUNICATION,
  CANSM_BSWM_SILENT_COMMUNICATION,
  CANSM_BSWM_FULL_COMMUNICATION,
  CANSM_BSWM_BUS_OFF
}
```

Listing 3.8: Modes reported by the API BswM_EthSM_CurrentState

```
modeGroup EthSM_NetworkModeStateType{
  ETHSM_UNINITED,
  ETHSM_NO_COMMUNICATION,
  ETHSM_FULL_COMMUNICATION
}
```

Listing 3.9: Modes reported by the API BswM_FrSM_CurrentState

```
modeGroup FrSM_BswM_StateType{
  FRSM_BSWM_READY,
  FRSM_BSWM_READY_ECU_PASSIVE,
  FRSM_BSWM_STARTUP,
  FRSM_BSWM_STARTUP_ECU_PASSIVE,
  FRSM_BSWM_WAKEUP,
  FRSM_BSWM_WAKEUP_ECU_PASSIVE,
  FRSM_BSWM_HALT_REQ,
  FRSM_BSWM_HALT_REQ_ECU_PASSIVE,
  FRSM_BSWM_KEYSLOT_ONLY,
  FRSM_BSWM_KEYSLOT_ONLY_ECU_PASSIVE,
  FRSM_BSWM_ONLINE,
  FRSM_BSWM_ONLINE_ECU_PASSIVE,
  FRSM_BSWM_ONLINE_PASSIVE,
  FRSM_BSWM_ONLINE_PASSIVE_ECU_PASSIVE
}
```

Listing 3.10: Modes reported by the API BswM_LinSM_CurrentState

```
modeGroup LinSM_ModeType{
  LINSM_FULL_COM,
  LINSM_NO_COM
}
```

Listing 3.11: Modes reported by the API BswM_EcuM_CurrentState

```
modeGroup EcuM_StateType{
  ECUM_SUBSTATE_MASK,
  ECUM_STATE_STARTUP,
  ECUM_STATE_STARTUP_ONE,
  ECUM_STATE_STARTUP_TWO,
  ECUM_STATE_WAKEUP,
  ECUM_STATE_WAKEUP_ONE,
  ECUM_STATE_WAKEUP_VALIDATION,
  ECUM_STATE_WAKEUP_REACTION,
  ECUM_STATE_WAKEUP_TWO,
  ECUM_STATE_WAKEUP_WAKESLEEP,
  ECUM_STATE_WAKEUP_TTII,
  ECUM_STATE_RUN,
  ECUM_STATE_APP_RUN,
  ECUM_STATE_APP_POST_RUN,
  ECUM_STATE_SHUTDOWN,
  ECUM_STATE_PREP_SHUTDOWN,
  ECUM_STATE_GO_SLEEP,
  ECUM_STATE_GO_OFF_ONE,
  ECUM_STATE_GO_OFF_TWO,
  ECUM_STATE_SLEEP,
```

```
  ECUM_STATE_RESET,
  ECUM_STATE_OFF
}
```

Listing 3.12: Modes reported by the API BswM_EcuM_CurrentWakeup

```
modeGroup EcuM_WakeupStatusType{
  ECUM_WKSTATUS_NONE,
  ECUM_WKSTATUS_PENDING,
  ECUM_WKSTATUS_VALIDATED,
  ECUM_WKSTATUS_EXPIRED,
  ECUM_WKSTATUS_DISABLED
}
```

Listing 3.13: Modes reported by the API BswM_NvM_CurrentBlockMode

```
modeGroup NvM_BlockMode {
  NVM_BLK_BLOCK_SKIPPED,
  NVM_BLK_INTEGRITY_FAILED,
  NVM_BLK_NOT_OK,
  NVM_BLK_NV_INVALIDATED,
  NVM_BLK_OK,
  NVM_BLK_PENDING,
  NVM_BLK_REDUNDANCY_FAILED,
  NVM_BLK_RESTORED_FROM_ROM
}
```

Listing 3.14: Modes reported by the API BswM_NvM_CurrentJobMode

```
modeGroup NvM_JobMode {
  NVM_JOB_CANCELED,
  NVM_JOB_NOT_OK,
  NVM_JOB_OK,
  NVM_JOB_PENDING
}
```

Listing 3.15: Modes reported by the API BswM_LinTp_RequestMode

```
modeGroup LinTp_Mode {
  LINTP_APPLICATIVE_SCHEDULE,
  LINTP_DIAG_REQUEST,
  LINTP_DIAG_RESPONSE
}
```

Listing 3.16: Modes reported by the API BswM_WdgM_RequestPartitionReset

```
modeGroup WdgM_PartitionResetType{
  WDGM_PARTITION_RESET_REQUESTED,
  WDGM_PARTITION_RESET_NOTREQUESTED

}
```

For the Diagnostic Communication Manager (DCM) there are two `ModeDeclarationGroup`s declared. Listing 3.17 shows the modes that determine which types communication are enabled or disabled during diagnostics. When the DCM wants to reset the ECU it has to indicated to the BswM which kind of reset should be executed. The various modes of reset can be seen in Listing 3.18.

Listing 3.17: Modes reported by the API BswM_DCM_CommunicationMode_CurrentState

```
modeGroup Dcm_CommunicationModeType{
  DCM_ENABLE_RX_TX_NORM,
  DCM_ENABLE_RX_DISABLE_TX_NORM,
  DCM_DISABLE_RX_ENABLE_TX_NORM,
  DCM_DISABLE_RX_TX_NORMAL,
  DCM_ENABLE_RX_TX_NM,
  DCM_ENABLE_RX_DISABLE_TX_NM,
  DCM_DISABLE_RX_ENABLE_TX_NM,
  DCM_DISABLE_RX_TX_NM,
  DCM_ENABLE_RX_TX_NORM_NM,
  DCM_ENABLE_RX_DISABLE_TX_NORM_NM,
  DCM_DISABLE_RX_ENABLE_TX_NORM_NM,
  DCM_DISABLE_RX_TX_NORM_NM
}
```

Listing 3.18: Modes reported by the API BswM_DCM_ResetMode_CurrentState

```
modeGroup Dcm_ResetModeType{
  DCM_NO_RESET,
  DCM_HARD_RESET,
  DCM_KEY_ON_OFF_RESET,
  DCM_SOFT_RESET,
  DCM_ENABLE_RAPID_POWER_SHUTDOWN_RESET,
  DCM_DISABLE_RAPID_POWER_SHUTDOWN_RESET,
  DCM_BOOTLOADER_RESET,
  DCM_SS_BOOTLOADER_RESET,
  DCM_RESET_EXECUTION
}
```

3.2.2.2 Exemplary ModeDeclarationGroups for this document

Listing 3.19: Application ModeDeclarationGroup

```
modeGroup App1Mode {
  APP1_ACTIVE,
  APP1_INACTIVE
  }
```

3.2.3 Definition of the interface in pseude code

3.2.3.1 Definition of ModeRequestPorts which are realized by the standardized interface of the BSWM

In the BSWM configuration, the mode request sources have to be defined. The following ModeRequestPorts are implicitly defined by API of the BSWM. This subsection summarizes the port interface.

3.2.3.1.1 BswMComMIndication

Purpose: Function called by ComM to indicate its current state.

Signature:

```
void BswM_ComM_CurrentMode(
        NetworkHandleType Network,
        ComM_ModeType RequestedMode
)
```

Definition:

```
request BswMComMIndication ComM_Mode_Channel1 {
  processing IMMEDIATE
  initialValue COMM_NO_COMMUNICATION
  source {Reference to [ComMChannel]}
}
```

Note: This ModeRequestSource has to be created once for each ComM-Channel.

3.2.3.1.2 BswMComMPncRequest

Purpose: Function called by ComM to indicate the current state of a partial network.

Signature:

```
void BswM_ComM_CurrentPNCMode(
        PNCHandleType PNC,
        ComM_ModeType RequestedMode
)
```

Definition:

```
request BswMComMPncRequest Pnc1Request {
  processing IMMEDIATE
  initialValue COMM_NO_COM_NO_PENDING_REQUEST
  source Reference to [ComMPnc]
}
```

Note: This ModeRequestSource has to be created once for each partial network.

3.2.3.1.3 BswMDcmComModeRequest

Purpose: Function called by DCM to indicate the current state of CommunicationControl.

Signature:

```
void BswM_Dcm_CommunicationMode_CurrentState(
        NetworkHandleType Network,
        Dcm_CommunicationModeType Mode
)
```

Definition:

```
request BswMDcmCommunicationMode
  BswM_Dcm_CommunicationMode_CurrentState {
  processing IMMEDIATE
  initialValue DCM_ENABLE_RX_TX_NORM
}
```

3.2.3.1.4 BswMDcmResetModeRequest

Purpose: Function called by DCM to indicate the current state of ResetMode.

Signature:
```
void BswM_Dcm_ResetMode_CurrentState(
        Dcm_ResetCtrlType Mode
)
```

Definition:
```
request BswMDcmResetMode BswM_Dcm_ResetMode_CurrentState {
  processing IMMEDIATE
  initialValue DCM_NO_RESET
}
```

3.2.3.1.5 BswMCanSMIndication

Purpose: Function called by CanSM to indicate its current state.

Signature:
```
void BswM_CanSM_CurrentState(
        NetworkHandleType Network,
        CanSM_BswMCurrentStateType CurrentState
)
```

Definition:
```
request BswMCanSMIndication CanSM_Can1 {
  processing IMMEDIATE
  initialValue CANSM_BSWM_NO_COMMUNICATION
  source Reference to [CanSMManagerNetwork]
}
```

Note: This ModeRequestSource has to be created once for each CAN channel.

3.2.3.1.6 BswMEthSMIndication

Purpose: Function called by EthSM to indicate its current state.

Signature:
```
void BswM_EthSM_CurrentState(
    NetworkHandleType Network,
    EthSM_NetworkModeStateType CurrentState
)
```

Definition:
```
request BswMEthSMIndication EthSM_Network1 {
  processing IMMEDIATE
  initialValue ETHSM_NO_COMMUNICATION
  source {Reference to [EthSmNetwork]}
}
```

Note: This ModeRequestSource has to be created once for each ethernet channel.

3.2.3.1.7 BswMFrSMIndication

Purpose: Function called by FrSM to indicate its current state.

Signature:
```
void BswM_FrSM_CurrentState(
        NetworkHandleType Network,
        FrSM_BswM_StateType CurrentState
)
```

Definition:
```
request BswMFrSMIndication FrSM_BswM_StateType {
  processing IMMEDIATE
  initialValue FRSM_BSWM_READY
  source {Reference to [FrSMCluster]}
}
```

Note: This ModeRequestSource has to be created once for each FlexRay cluster.

3.2.3.1.8 BswMLinSMIndication

Purpose: Function called by LinSM to indicate its current state.

Signature:
```
void BswM_LinSM_CurrentState(
    NetworkHandleType Network,
    LinSM_ModeType CurrentState
)
```

Definition:
```
request BswMLinSMIndication LinSM_CurrentState {
  processing IMMEDIATE
  initialValue LINSM_NO_COM
  source {Reference to [LinSMChannel]}
}
```

Note: This ModeRequestSource has to be created once for each Lin channel.

3.2.3.1.9 BswMEcuMIndication

Purpose: Function called by the ECUM with fixed state machine to indicate its current state.

Signature:
```
void BswM_EcuM_CurrentState(
    NetworkHandleType Network,
    LinSM_ModeType CurrentState
)
```

Definition:
```
request BswMEcuMIndication EcuM_State {
  processing IMMEDIATE
  initialValue ECUM_STATE_STARTUP
}
```

3.2.3.1.10 BswMEcuMWakeupSource

Purpose: Function called by the ECUM to indicate the current state of the wakeup sources.

Signature:

```
void BswM_EcuM_CurrentWakeup(
    EcuM_WakeupSourceType source,
    EcuM_WakeupStatusType state
)
```

Definition:

```
request BswMEcuMWakeupSource EcuM_WakeupSource {
  processing IMMEDIATE
  initialValue ECUM_WKSTATUS_NONE
  source {Reference to [EcuMWakeupSource]}
}
```

Note: This ModeRequestSource has to be created once for each Wakeup source.

3.2.3.1.11 BswMLinScheduleIndication

Purpose: Function called by LinSM to indicate the currently active schedule table for a specific LIN channel.

Signature:

```
void BswM_LinSM_CurrentSchedule(
        NetworkHandleType Network,
        LinIf_SchHandleType CurrentSchedule
)
```

Definition:

```
request BswMLinScheduleIndication LinSM1_CurrentSchedule {
  processing IMMEDIATE
  initialValue TBD
  source {Reference to [LinSMSchedule]}
}
```

3.2.3.1.12 BswMLinTpModeRequest

Purpose: Function called by LinTP to request a mode for the corresponding LIN channel. The LinTp_Mode mainly correlates to the LIN schedule table that should be used.

Signature:

```
void BswM_LinTp_RequestMode(
        NetworkHandleType Network,
        LinTp_Mode LinTpRequestedMode
)
```

Definition:

```
request BswMLinTpModeRequest LinTp_Mode {
  processing IMMEDIATE
  initialValue LINTP_APPLICATIVE_SCHEDULE
  source {Reference to [ LinIfChannel ]}
```

```
}
```

3.2.3.1.13 BswMWdgMRequestPartitionReset

Signature:

```
void BswM_WdgM_RequestPartitionReset(
        ApplicationType Application
)
```

Definition:

```
request BswMWdgMRequestPartitionReset WdgM_RequestResetPart1
  {
  processing IMMEDIATE
  initialValue WDGM_PARTITION_RESET_NOTREQUESTED
  source  {Reference to [EcucPartition]}
}
```

Note: This ModeRequestSource has to be created once for each partition for which a reset can be requested by the Watchdog Manager module.

3.2.3.2 Definition of configurable ModeRequestPorts

Besides the interface, which is defined by the standardized interface of the BSWM, additional mode request ports can be defined via the configuration parameters.

E.g it is necessary for the interaction with applications, that an application software component at least notifies the BSWM about it's current state. This can be achieved by definition of a `ModeRequestPort` as shown in Listing 3.20. The BSWM will than create a corresponding `RPort` typed by a `SenderReceiverInterface`.

Listing 3.20: Application `ModeRequestPort`

```
request SwcModeRequest App1ModeRequest {
  type App1Mode // Reference to ModeDeclarationGroupPrototype
  processing IMMEDIATE
  initialValue "APP1_INACTIVE"
}
```

Note that the reference to a `ModeDeclarationGroupPrototype` can be misleading. The meaning is that the BSWM creates a `SenderReceiverInterface` containing a `VariableDataPrototype`. The `SwDataDefProps` of this `VariableDataPrototype` refer to a `CompuMethod`, which defines an enumeration corresponding die to the referred `ModeDeclarationGroupPrototype`.

Listing 3.21: Application `ModeNotification`

```
request SwcModeNotification App1ModeNotification {
  type App1Mode //Reference to ModeDeclarationGroupPrototype
  processing IMMEDIATE
  initialValue "APP1_INACTIVE"
}
```

Listing 3.21 shows the declaration of a mode notification port. Note that in contrast to 3.20 the BSWM will generate a `RPort` typed by a `ModeSwitchInterface` in this case. The BSWM then gets informed via a `ModeSwitchNotification` if the mode manager initiates a mode switch.

Listing 3.22: `BasicSoftwareModeNotification`

```
request BswModeNotification BswMode1 {
  type BswMode //Reference to ModeDeclarationGroupPrototype
  processing IMMEDIATE
  initialValue "BSW_INACTIVE"
}
```

Listing 3.22 shows the declaration of a mode notification port. If such a port is configured, the BSWM configuration tool will create a `requiredModeGroup` `ModeDeclarationGroupPrototype`, so that the BSWM gets informed of mode switches via the Schedule Manager, if the corresponding mode manager initiates a mode switch with a call to `SchM_Switch` API.

3.2.4 Configuration of the BSWM behavior

The behavior of the BSWM is specified via rules and action lists. A rule is a logical expression, which combines the current values of `ModeRequestPort`s. The evaluation of each rule either results in the execution of its *true* or *false* action lists.

The `ModeControlContainer` contains these `ActionLists`. An `ActionList` can consist of a set of atomic actions, other “nested” `ActionLists` or it can reference (nested) rules which are then evaluated in the context of this `Actionlist`.

The following example shows a simple rule, which activates the IPDU Groups of a dedicated CAN channel. According to this rule, the BSWM has to provide a `ModeRequestPort` of type `CanSMIndication` named `Can1_Indication`. This is a `ModeRequest` from a basic software module in this case from the Can State manager. In code this `ModeRequestPorts` corresponds to the API `BswM_CanSM_CurrentState` as described in BSWM0049 in [5]. The `source` parameter identifies the network to which this `ModeRequestSourcePort` belongs to. It's up to the configuration tool of the BSWM to allocate the right parameters for the API corresponding to the referenced ECUC Container.

The value of the `ModeRequestSourcePort` initially is `CAN_SM_BSWM_NO_COMMUNICATION`.

`processing immediate` means that every evaluation rule, which refers to this `ModeRequestSourcePort` shall immediately be processed. If this parameter would be `deferred` in case of a ModeRequest, the evaluation of rules would be delayed until the next run of the main function of the BSWM.

The following example shows an arbitration rule called `canIPDUActivation`. The overall content is rather self explanatory. The `initial` parameters specifies that the initial result of the rule evaluation is `false`.

Listing 3.23: Example for a rule

```
canIPDUActivation initial false on TRIGGER {
  if (  Can1_indication == FULL_COM )
  {
    activateCANPDUs
  } else {
    deactivateCANPDUs
  }
}
```

At which point in time a rule is executed, after an event has occurred depends on the parameter `BswMActionListExecution`. Either it is executed every time the rule is evaluated with the corresponding result, or only when the evaluation result has changed from the previous evaluation. This is called `triggered` respectively `conditional` execution.

Table 3.1 gives an overview in which situations an `ActionList` is executed or not. Triggered `ActionLists` are executed (triggered) if the result of the rule evaluation changes. Conditional `ActionLists` depend only on the current result (condition) of the evaluation independent if it has changed or not.

Table 3.1: Execution of Action Lists depending on parameter `BswMActionListExecution`

eval. result (*old*) -> (*new*)	*true -> true*	*true -> false*	*false -> false*	*false -> true*
TrueActionList	CONDITION	-	-	TRIGGERED/ CONDITION
FalseActionList	-	TRIGGERED/ CONDITION	CONDITION	-

3.3 ECU State management

During startup and shutdown the task of the BSWM is to initialize all basic software modules in a similar way as it is done by the ECUM in older AUTOSAR releases.

3.3.1 Startup

The ECUM starts the operating system and initializes in its *post OS sequence* the Schedule manager and the BSWM. The BSWM then has to take care, that all necessary init routines of the basic software modules are called and that the RTE is started.

In this scenario it is expected that the BSWM has the following *providedModeDeclarationGroup*. The purpose of this ModeDeclarationGroup is to track the current state/-mode of the ECU similar to the states of the ECU State manager in previous AUTOSAR releases.

Rule *InitBlockII* specifies the initialization of basic drivers to access the NVRAM and initiates *NvM_ReadAll*. As the EcuMode source has the processing attribute set to

DEFERRED this rule will be evaluated every time the main function of the BSWM is called. After the first run it sets the EcuMode to *ECUM_STATE_STARTUP_TWO* so that the action list will never be invoked again.

If the *NvMReadAll* job is finsihed the *NvMReadAllFinished* rule is triggered, which initiates the remaining initialization and switches the EcuMode to *ECUM_STATE_RUN*.

Listing 3.24: Rules and ActionLists for Startup

```
rule InitBlockII initial false on CONDITION {
  if (EcuMode == ECUM_STATE_STARTUP_ONE )
  {
    custom "Spi_Init(null)"
    custom "Eep_Init(null)"
    custom "Fls_Init(null)"
    custom "NvM_Init(null)"
    SchMSwitch(EcuMode,ECUM_STATE_STARTUP_TWO)
    custom "NvM_ReadAll()"
  }
}

rule NvMReadAllFinished initial false on TRIGGER {
  if (NvMReadAllJobMode == NVM_JOB_FINISHED && EcuMode ==
   ECUM_STATE_STARTUP_TWO) {
    custom "Can_Init(null)"
    custom "CanIf_Init(null)"
    custom "CanSM_Init(null)"
    custom "CanTp_Init(null)"
    custom "Lin_Init(null)"
    custom "LinIf_Init(null)"
    custom "LinSM_Init(null)"
    custom "LinTp_Init(null)"
    custom "Fr_Init(null)"
    custom "FrIf_Init(null)"
    custom "FrSM_Init(null)"
    custom "FrTp_Init(null)"
    custom "PduR_Init(null)"
    custom "CANNM_Init(null)"
    custom "FrNM_Init(null)"
    custom "NmIf_Init(null)"
    custom "IpduM_Init(null)"
    custom "COM_Init(null)"
    custom "DCM_Init(null)"
    custom "ComM_Init(null)"
    custom "DEM_Init(null)"
    custom "StartRte()"
    SchMSwitch(EcuMode,ECUM_STATE_RUN)
  }
}
```

When the RTE is started the runnables will be started. Now it is up to the application to keep the ECU running. To achieve this the BSWM can for example provide a *ModeRequestPort* as depicted in example 3.20. For the further reading is is expected, that the application software requests the mode *APP1_ACTIVE* from the BSWM. If this mode is requested the BSWM shall not shutdown the ECU.

Listing 3.25: Application runs, enable communication

```
rule checkApp1Request initial false on TRIGGER {
  if (App1Mode == APP1_ACTIVE && EcuMode == ECUM_STATE_RUN) {
    CommunicationAllowed CanSM.CanStateManagerConfiguration.
      Can1StateManagerNetwork true
    SchMSwitch(EcuMode,ECUM_STATE_APP_RUN)

  }
}
```

3.3.2 Run

As the BSWM is a highly flexible module it depends to a high extend to the integrator, how it is determined if an ECU shall shutdown or not. Many different variants are conceivable. This document proposes an approach, which is quite similar to the concept of the ECUM in AUTOSAR R3.1. The general concept is, that a ECU keeps running as long as at least one application software component requests the run state.

The information if an application can be shut down in a certain mode has to be provided by the software component developer. Example 3.26 shows a simplified rule for an ECU with one software component. If switches its mode to *INACTIVE* the BSWM initiates the shutdown sequence.

Listing 3.26: Initiate shutdown, if no application wants to run any more

```
rule checkApp1Request initial false on TRIGGER {
  if (App1Mode == APP1_INACTIVE && EcuMode == ECUM_STATE_RUN) {
    ComMCommunicationAllowed CanSM.CanStateManagerConfiguration.
      Can1StateManagerNetwork false
    SchMSwitch(EcuMode,ECUM_STATE_APP_POST_RUN)
  }
}
```

3.3.3 Shutdown

In state *ECUM_STATE_APP_POST_RUN* the BSWM waits until all channels report, that no requests are pending any more. The rule in listing 3.26 is triggered every time the mode of a ComM channel changes. If there are more than one ComM channel, they have to be combined to a single expression.

Listing 3.27: Shutdown sequence

```
rule InitiateShutdown initial false on TRIGGER {
  if (ComM_Mode_Channel1 == COMM_NO_COM_REQUEST_PENDING && EcuMode ==
    ECUM_STATE_APP_POST_RUN)
  {
    custom "Dem_Shutdown(null)"
    custom "Rte_Stop()"
    custom "ComM_DeInit()"
```

```
    SchMSwitch(EcuMode,ECUM_STATE_PREP_SHUTDOWN)
    custom "NvM_WriteAll()"
  }
}

rule NvMWriteAllFinished initial false on TRIGGER
{
  if (NvMWriteAllJobMode == NVM_JOB_FINISHED && EcuMode ==
    ECUM_STATE_PREP_SHUTDOWN)
  {
    custom "EcuM_GoDown(MODULE_ID)" // MODULE_ID of BSWM: 42
  }
}
```

Note that the in the Configuration of the ECUM the module id of the BSWM has to be added as a valid user to `EcuMFlexUserConfig`.

3.3.4 Sleep

Entering a sleep state is similar to the shutdown sequence 3.26 except that *EcuM_GoHalt()* resp. *EcuM_GoPoll()* is called instead of *EcuM_GoDown*.

3.3.5 Wakeup

Example 3.28 shows a rule which starts the ECU only, if a certain wakeup event, identified by *EcuM_WakeupSource* has occured. Otherwise the ECU will be immediately shut down.

Listing 3.28: start sequence with wakeup check

```
rule InitBlockII initial false on CONDITION {
  if (EcuMode == ECUM_STATE_STARTUP_ONE && EcuM_WakeupSource ==
    ECUM_WKSTATUS_VALIDATED)
  {
    custom "Spi_Init(null)"
    custom "Eep_Init(null)"
    custom "Fls_Init(null)"
    custom "NvM_Init(null)"
    SchMSwitch(EcuMode,ECUM_STATE_STARTUP_TWO)
    custom "NvM_ReadAll()"
  } else {
    custom "EcuM_GoDown()"
  }
}
```

3.4 Communication Management

Besides parts of the ECU state management, a part of the communication management is in the responsibility of the BSWM. This section describes the functionality of the BSWM, which is related to the Communication Stack of AUTOSAR. This covers but is not restricted to the following uses cases.

- Starting and stopping of IPDU Groups in general
- Partial Networking
- Diagnostic use cases which influence the communication of an ECU. e.g. it might be necessary to set the FlexRay State manager to passive mode via *FrSm_SetEcuPassive()* when requested by an application.

To fulfill the requested functionality the BSWM has ModeRequestSources to

- the Communication Manager
- the bus state managers
- AUTOSAR COM

3.4.1 Startup of ECU

Besides the initialization of the communication stack the BswM can be configured to initialize further modules or execute customs actions depending on the ECU's needs. Due to the flexibility of the BSWM it is also possible, that after a wake up event only a part of the communication stack is started.

3.4.2 Shutdown of ECU

Analogue to Startup, it is possible to configure additional actions to be executed on shutdown.

3.4.3 I-PDU Group Switching

For the I-PDU group switching there exists for each channel a dedicated I-PDU group for outgoing and incoming I-PDUs. AUTOSAR COM takes care that an I-PDU is active(started) if at least one I-PDU group containing this I-PDU is active.

To illustrate how the I-PDUs of an ECU can be managed the following scenario is created. The examplary ECU shall have two CAN channels and three partial networks. The mode request ports for the channels are named *Channel1* and *Channel1*, the request sources for the partial networks are named *PNC1*, *PNC2* and *PNC3*.

I-PDUs of *PNC1* shall be communicated only over *Channel1*. I-PDUs of *PNC3* shall be communicated over *Channel1* and *Channel2*. I-PDUs of *PNC3* shall be communicated only over *Channel2*.

Listing 3.29: Active wakeup on channel

```
rule activeWakeupChannel1 initial false on CONDITION {
  if (CanSMChannel1 == CANSM_BSWM_FULL_COMMUNICATION &&
    PNC1 != PNC_REQUESTED &&
    PNC2 != PNC_REQUESTED
    )
  {
    PduGroupSwitch {init = true,
            enable CAN1IPDUS
            }
  }
}

rule activeWakeupChannel2 initial false on CONDITION {
  if (CanSMChannel2 == CANSM_BSWM_FULL_COMMUNICATION &&
    PNC2 != PNC_REQUESTED &&
    PNC3 != PNC_REQUESTED
    )
  {
    PduGroupSwitch {init = true,
            enable CAN2IPDUS
            }
  }
}
```

Listing 3.30: CanSM reports SILENT_COMMUNICATION or NO_COMMUNICATION

```
rule stopComChannel1 initial false on CONDITION {
  if (CanSMChannel1 == CANSM_BSWM_SILENT_COMMUNICATION ||
    CanSMChannel1 == CANSM_BSWM_NO_COMMUNICATION
    )
  {
    PduGroupSwitch {init = true,
            disable  CAN1IPDUS, PNC1IPDUS, PNC2IPDUS
            }
  }
}

rule stopChannel2 initial false on CONDITION {
  if (CanSMChannel2 == CANSM_BSWM_SILENT_COMMUNICATION ||
    CanSMChannel2 == CANSM_BSWM_NO_COMMUNICATION
    )
  {
    PduGroupSwitch {init = true,
            disable = CAN2IPDUS, PNC2IPDUS, PNC3IPDUS
            }
  }
}
```

Listing 3.31: PNC reports NO_COMMUNICATION

```
rule pnc1nocom initial false on TRIGGER {
```

```
  if (PNC1 == PNC_NO_COMMUNICATION )
  {
    PduGroupSwitch {init = false,
            disable PNC1IPDUS
            }
    DeadlineMonitoringControl {
            disable PNC1IPDUS
            }
  }
}

rule pnc2nocom initial false on TRIGGER {
  if (PNC2 == PNC_NO_COMMUNICATION )
  {
    PduGroupSwitch {init = false,
            disable PNC2IPDUS
            }
    DeadlineMonitoringControl {
            disable PNC2IPDUS
            }
  }
}

rule pnc3nocom initial false on TRIGGER {
  if (PNC3 == PNC_NO_COMMUNICATION )
  {
    PduGroupSwitch {init = false,
            disable PNC3IPDUS
            }
    DeadlineMonitoringControl {
            disable PNC3IPDUS
            }
  }
}
```

Listing 3.32: PNC reports PNC_REQUESTED or PNC_READY_SLEEP

```
rule pnc1requested initial false on TRIGGER {
  if (PNC1 == PNC_REQUESTED ||
    PNC1 == PNC_READY_SLEEP )
  {
    PduGroupSwitch {init = false,
            enable PNC1IPDUS
            }
  }
}

rule pnc2requested initial false on TRIGGER {
  if (PNC2 == PNC_REQUESTED ||
    PNC2 == PNC_READY_SLEEP )
  {
    PduGroupSwitch {init = false,
            enable PNC2IPDUS
            }
  }
}
```

```
rule pnc3requested initial false on TRIGGER {
  if (PNC3 == PNC_REQUESTED ||
    PNC3 == PNC_READY_SLEEP )
  {
    PduGroupSwitch {init = false,
            enable PNC3IPDUS
            }
  }
}
```

Listing 3.33: PNC reports PNC_PREPARE_SLEEP

```
rule pnc1preparesleep initial false on TRIGGER {
  if (PNC1 == PNC_PREPARE_SLEEP)
  {
    PduGroupSwitch {
            init = false,
            enable PNC1IPDUS
            }
    DeadlineMonitoringControl {
            disable PNC1IPDUS
            }
  }
}

rule pnc2preparesleep initial false on TRIGGER {
  if (PNC2 == PNC_PREPARE_SLEEP )
  {
    PduGroupSwitch {init = false,
            enable PNC2IPDUS
            }
    DeadlineMonitoringControl {
            disable PNC2IPDUS
            }
  }
}

rule pnc3preparesleep initial false on TRIGGER {
  if (PNC3 == PNC_PREPARE_SLEEP )
  {
    PduGroupSwitch {init = false,
            enable PNC3IPDUS
            }
    DeadlineMonitoringControl {
            disable PNC3IPDUS
            }
  }
}
```

3.5 Diagnostics

In AUTOSAR release 4.0.3 onwards the DCM is the overall mode manager for all diagnostic use cases. The BSWM is responsible to change the state of the other basic software modules accordingly. The first use case is diagnostic communication control. If the DCM reports to the BSWM that a specified communication control mode is entered, the BSWM has to enable resp. disable the corresponding IPDU groups.

Listing 3.34 shows how this can be achievied via configuration of the BSWM.

Listing 3.34: CommunicationControl

```
rule communicationcontrol1 initial false on CONDITION {
  if (Dcm_Communication_Control_CAN1 == DCM_ENABLE_RX_TX_NORM )
  {
    PduGroupSwitch {init = false,
            enable CAN1IPDUS
            }
  }
}

rule communicationcontrol2 initial false on CONDITION {
  if (Dcm_Communication_Control_CAN1 == DCM_ENABLE_RX_DISABLE_TX_NORM )
  {
    PduGroupSwitch {init = false,
            enable CAN1RXIPDUS
            disable CAN1TXIPDUS
            }
  }
}

rule communicationcontrol3 initial false on CONDITION {
  if (Dcm_Communication_Control_CAN1 == DCM_DISABLE_RX_ENABLE_TX_NORM )
  {
    PduGroupSwitch {init = false,
            enable CAN1TXIPDUS
            disable CAN1RXIPDUS
            }
  }
}

rule communicationcontrol5 initial false on CONDITION {
  if (Dcm_Communication_Control_CAN1 == DCM_DISABLE_RX_TX_NORMAL )
  {
    PduGroupSwitch {init = false,
            disable CAN1IPDUS
            }
  }
}

rule communicationcontrol6 initial false on CONDITION {
  if (Dcm_Communication_Control_CAN1 == DCM_ENABLE_RX_TX_NM )
  {
    PduGroupSwitch {init = false,
            enable CAN1NMIPDUS
            }
```

```
  }
}

rule communicationcontrol7 initial false on CONDITION {
  if (Dcm_Communication_Control_CAN1 == DCM_ENABLE_RX_DISABLE_TX_NM )
  {
    PduGroupSwitch {init = false,
            enable CAN1NMRXIPDUS
            disable CAN1NMTXIPDUS
            }
  }
}

rule communicationcontrol8 initial false on CONDITION {
  if (Dcm_Communication_Control_CAN1 == DCM_DISABLE_RX_ENABLE_TX_NM )
  {
    PduGroupSwitch {init = false,
            enable CAN1NMTXIPDUS
            disable CAN1NMRXIPDUS
            }
  }
}

rule communicationcontrol9 initial false on CONDITION {
  if (Dcm_Communication_Control_CAN1 == DCM_DISABLE_RX_TX_NM )
  {
    PduGroupSwitch {init = false,
            disable CAN1NMRXIPDUS,  CAN1NMTXIPDUS
            }
  }
}

rule communicationcontrol10 initial false on CONDITION {
  if (Dcm_Communication_Control_CAN1 == DCM_ENABLE_RX_TX_NORM_NM )
  {
    PduGroupSwitch {init = false,
            enable CAN1NMRXIPDUS, CAN1NMTXIPDUS
            }
  }
}

rule communicationcontrol11 initial false on CONDITION {
  if (Dcm_Communication_Control_CAN1 == DCM_ENABLE_RX_DISABLE_TX_NORM_NM )
  {
    PduGroupSwitch {init = false,
            enable CAN1NMRXIPDUS, CAN1RXIPDUS
            disable CAN1NMTXIPDUS, CAN1TXIPDUS
            }
  }
}

rule communicationcontrol12 initial false on CONDITION {
  if (Dcm_Communication_Control_CAN1 == DCM_DISABLE_RX_ENABLE_TX_NORM_NM )
  {
    PduGroupSwitch {init = false,
            enable CAN1NMTXIPDUS, CAN1TXIPDUS
```

```
            disable CAN1NMRXIPDUS, CAN1RXIPDUS
            }
  }
}

rule communicationcontrol13 initial false on CONDITION {
  if (Dcm_Communication_Control_CAN1 == DCM_DISABLE_RX_TX_NORM_NM )
  {
    PduGroupSwitch {init = false,
            disable CAN1NMTXIPDUS,CAN1TXIPDUS, CAN1NMRXIPDUS,CAN1RXIPDUS
            }
  }
}
```

If the DCM has entered the reset mode it is up to the BSWM to execute the reset of the ECU immediately.

Listing 3.35: ResetMode

```
rule dcmsessioncontrol initial false on CONDITION {
  if (Dcm_ResetMode == DCM_HARD_RESET )
  {
    custom "Mcu_reset()"
  }
}
```

4 Backward Compatibility

This chapter describes a setup to reuse software components (legacy SWCs), which are designed to work with the "ECU State Manager (EcuM) with fixed state machine" [1]. This means that a setup based on EcuM with flexible state machines will be described which emulates the behavior of the EcuM with a fixed state machine.

An Overview of the approach is shown in Figure 4.1. A new Software Component *EcuM Fixed Compatibility SWC* is added to build a wrapper that presents an interface of an EcuM with a fixed state machine to the legacy SWCs.

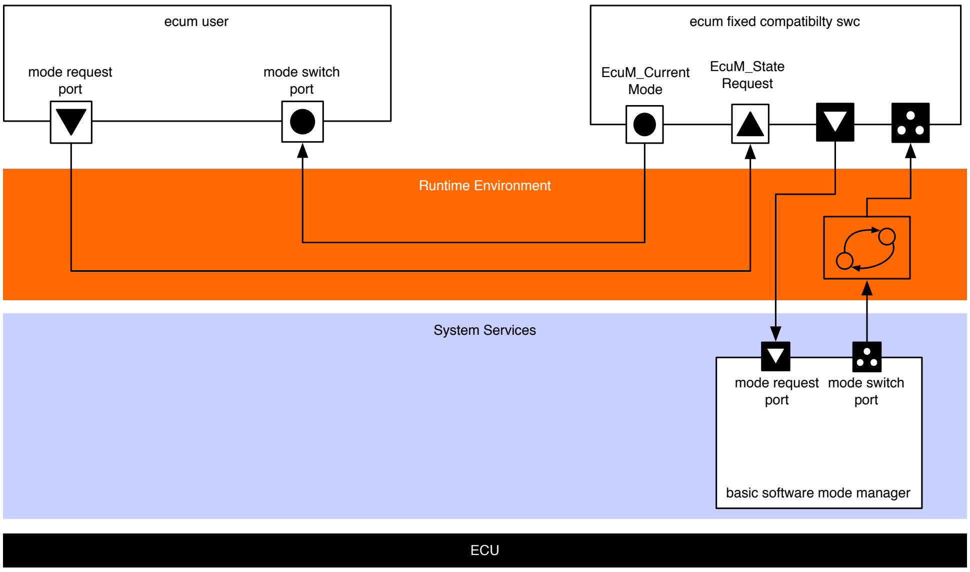

Figure 4.1: Use of SWCs designed to work with ECU State Manager with fixed state machine

Figure 4.2 shows the mapping from fixed EcuM to flexible EcuM. The small boxes represent the states of fixed EcuM and are sometimes included into green boxes which represent the phases of flexible EcuM. Every state of the fixed EcuM which is inside a green box of the flexible EcuM does not have to emulated because its execution is already included in the flexible EcuM. That leads to the necessity to emulate all states of the fixed EcuM that are not included in green boxes using the BswM during the UP phase of the flexible EcuM. This mapping of the states of fixed EcuM to the phases of the flexible EcuM is shown because the lifecycle of an ECU has changed massively from fixed EcuM to flexible EcuM. The lifecycle of the fixed EcuM corresponds to a AUTOSAR 3 based ECU.

So the BswM helps to emulate the fixed EcuM. For a backward compatible configuration the BswM must be configured in such a way that it executes these actions.

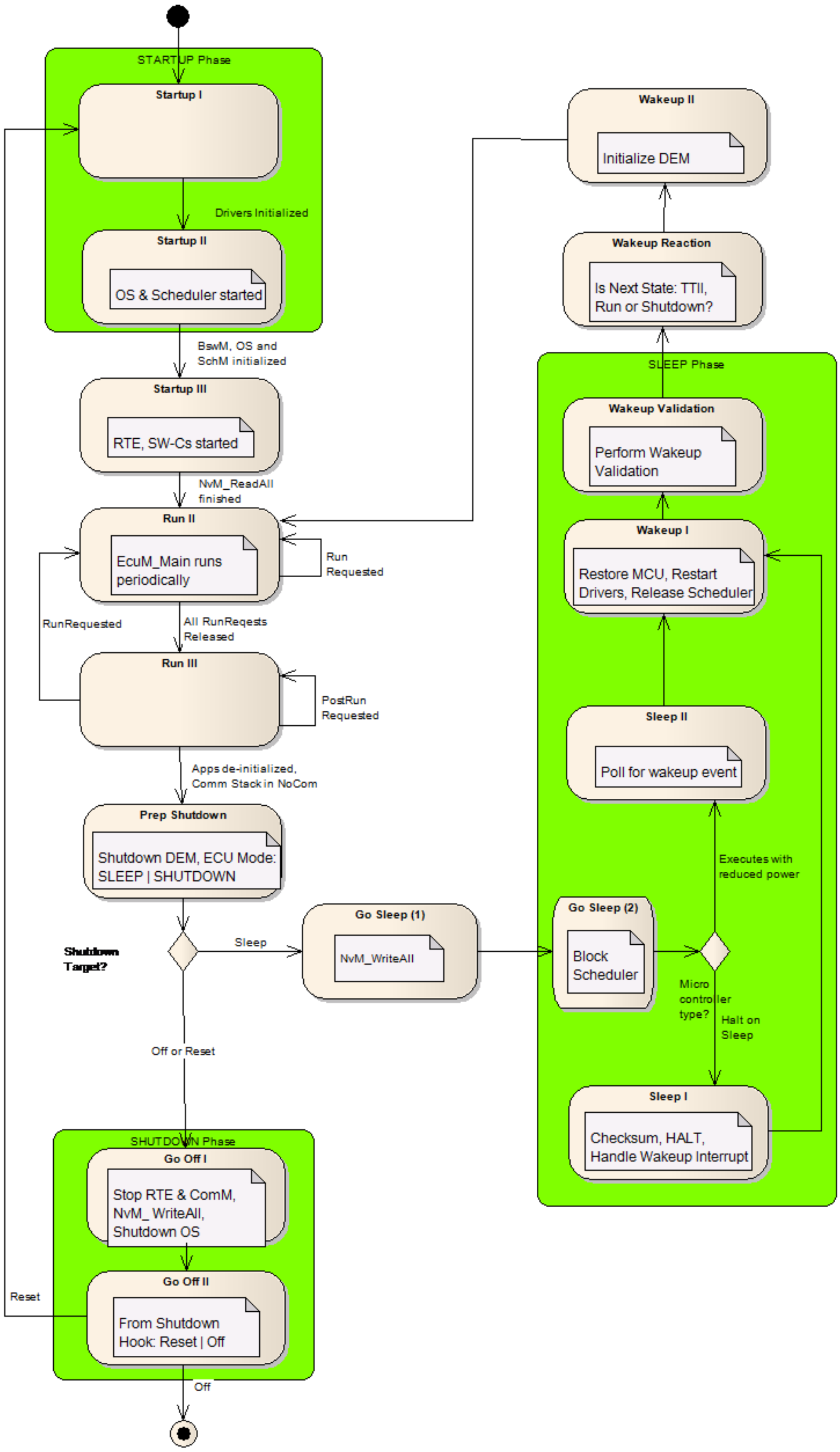

Figure 4.2: Mapping: Phases of fixed EcuM to flexible EcuM

This chapter describes a compatibility SWC and the modifications of the BswM configuration that are necessary. The following hints in this chapter for achieving backward compatibility are aligned along the phases of execution.

As most parts of the achievement of compatibility is done via BswM rules, this chapter shows only additional BswM rules and the modifications of the already introduced rules of chapter 3.

4.1 Startup

During startup phase the same BSW modules shall be initialized as the fixed EcuM does. This is implemented via BswM rules which are executed after initialization of EcuM and initialize these modules. The modules which are already initialized by flexible EcuM are omitted by BswM.

The changed BswM rules can be seen in Listing 4.1.

Listing 4.1: BswM configuration for fixed EcuM compatible startup

```
rule InitBlockII initial false on CONDITION {
  if (EcuMode == ECUM_STATE_STARTUP_ONE )
  {
    custom "EcuMCompatibility_SetStartup(null)"
    custom "Port_Init(null)"
    custom "Dio_Init(null)"
    custom "Adc_Init(null)"
    custom "Spi_Init(null)"
    custom "Eep_Init(null)"
    custom "Fls_Init(null)"
    custom "NvM_Init(null)"
    SchMSwitch(EcuMode,ECUM_STATE_STARTUP_TWO)
    custom "NvM_ReadAll()"
  }
}

rule NvMReadAllFinished initial false on TRIGGER {
  if (NvMReadAllJobMode == NVM_JOB_FINISHED && EcuMode ==
    ECUM_STATE_STARTUP_TWO) {
    custom "CanTrcv_Init(null)"
    custom "Can_Init(null)"
    custom "CanIf_Init(null)"
    custom "CanSM_Init(null)"
    custom "CanTp_Init(null)"
    custom "Lin_Init(null)"
    custom "LinIf_Init(null)"
    custom "LinSM_Init(null)"
    custom "LinTp_Init(null)"
    custom "FrTrcv_Init(null)"
    custom "Fr_Init(null)"
    custom "FrIf_Init(null)"
    custom "FrSM_Init(null)"
    custom "FrTp_Init(null)"
    custom "PduR_Init(null)"
```

```
      custom "CANNM_Init(null)"
      custom "FrNM_Init(null)"
      custom "NmIf_Init(null)"
      custom "IpduM_Init(null)"
      custom "COM_Init(null)"
      custom "DCM_Init(null)"
      custom "EcuMCompatibility_OnRteStartup()"
      custom "StartRte()"
      custom "ComM_Init(null)"
      custom "DEM_Init(null)"
      custom "FIM_Init(null)"
      custom "EcuMCompatibility_SetUp(null)"
      SchMSwitch(EcuMode,ECUM_STATE_RUN)
  }
}
```

4.2 Running

If the running phase is active, it is necessary for compatibility to emulate the interfaces of fixed EcuM as these are used by the legacy SWCs. There are two categories of interfaces: Those for getting the current mode and those for requesting a mode.

Firstly, in fixed EcuM the SWCs can get the current mode through the method *EcuM_CurrentMode()*. In this setup for compatibility, the legacy SWC does not use the method of EcuM but calls another method with the same name of the newly introduced EcuM Compatibility SWC which represents the wrapper. It gets the current mode of the flexible EcuM and transforms it into a mode of fixed EcuM which is known by the legacy components.

The mapping of the flexible ECU modes into fixed modes can be found in Table 4.1.

Table 4.1: Mapping of modes from flexible EcuM to fixed EcuM

Flexible EcuM	Fixed EcuM
ECUM_STATE_STARTUP_ONE	STARTUP
ECUM_STATE_STARTUP_TWO	STARTUP
ECUM_STATE_RUN	RUN
ECUM_STATE_APP_RUN	RUN
ECUM_STATE_APP_POST_RUN	POST_RUN
ECUM_STATE_GoSleep	SLEEP
ECUM_STATE_SleepWaitForNvMWriteAll	SLEEP
ECUM_STATE_GoOff1	SHUTDOWN
ECUM_STATE_GoOff2	SHUTDOWN

Secondly, legacy SWCs have to be able to request modes. Analogue to the approach sketched above, the legacy components to not communicate directly with the EcuM but with the compatibility SWC. The compatibility SWC offers the same interfaces as fixed EcuM and relays the request to flexible EcuM. The interface is called *EcuM_ModeRequest* and its methods to emulate are: *EcuM_RequestRUN()*, *EcuM_ReleaseRUN()*, *EcuM_RequestPOST_RUN()*, *EcuM_ReleasePOST_RUN()* and *EcuM_KillAllRunRequests()*.

For *each* fixed EcuM User the compatibility component needs an own *EcuM_ModeRequest*-Port as this would also be provided by fixed EcuM. The legacy SWC then gets connected to exactly that port which belongs to the requested user.

The needed configuration of BswM is shown in Listing 4.2. This includes the declaration of a mode group which represents the requested mode. This information is given to the BswM. The shown rule is responsible for activating the communication if running mode was requested.

Listing 4.2: BswM configuration for fixed EcuM compatible running mode

```
modeGroup EcuMCompatibilityMode {
  ECUMCOMPATIBILITY_Run,
  ECUMCOMPATIBILITY_PostRun
  ECUMCOMPATIBILITY_Off,
}

rule checkEcuMCompatibilityModeRequest initial false on TRIGGER {
  if (EcuMCompatibilityMode == ECUMCOMPATIBILITY_Run && EcuMode ==
    ECUM_STATE_RUN) {
     CommunicationAllowed CanSM.CanStateManagerConfiguration.
       Can1StateManagerNetwork true
     SchMSwitch(EcuMode,ECUM_STATE_APP_RUN)
  }
}
```

The compatibility SWC has to take all requested modes of the legacy components and transform all requests into *one* mode which is given to the BswM. This consists of two steps:

1. Depending on the called wrapping-method choose a mode of the above stated flexible EcuM modes. The mapping is described in Table 4.2.
2. Determine the “highest” mode of all compatibility users and request that from the BswM where *ECUMCOMPATIBILITY_Run* has the highest priority and *ECUMCOMPATIBILITY_Off* has the lowest.

Table 4.2: Mapping of modes requests from flexible EcuM to fixed EcuM

Called Method	Mode
EcuM_RequestRUN()	ECUMCOMPATIBILITY_Run
EcuM_ReleaseRUN()	ECUMCOMPATIBILITY_Off

EcuM_RequestPOST_RUN()	ECUMCOMPATIBILITY_PostRun
EcuM_ReleasePOST_RUN()	ECUMCOMPATIBILITY_Run
EcuM_KillAllRunRequests()	ECUMCOMPATIBILITY_Off

If the method *EcuM_KillAllRunRequests()* is called, the compatibility component requests *ECUMCOMPATIBILITY_Off* from the BswM independent of other legacy SWC's requests.

4.3 Shutdown

If no legacy SWC requested the running mode, the compatibility SWC signals that to the BswM via the mode *ECUMCOMPATIBILITY_Off* and BswM can decide whether it wants to keep the ECU running, shut it down or put it into sleep. If it shall be shut down or put into sleep, the BswM goes to post-run phase. During post-run phase a new request can bring the BswM into running mode again.

For that shutdown mechanism the BswM configuration of Listing 4.3 is responsible. The listed rules coordinate the post-run phase, deinitialize the modules and put the ECU into shut down or sleep. These rules execute the same callouts *EcuM_On<Mode>()* as it would happen with a fixed EcuM. As the callouts during startup and shutdown cannot be called by the compatibility SWC, they are executed by the BswM via `custom calls`.

Listing 4.3: BswM configuration for fixed EcuM compatible shutdown

```
rule checkEcuMCompatibilityModeRequest initial false on TRIGGER {
  if (EcuMCompatibilityMode != ECUMCOMPATIBILITY_Run && EcuMode ==
    ECUM_STATE_APP_RUN) {
    ComMCommunicationAllowed CanSM.CanStateManagerConfiguration.
      Can1StateManagerNetwork false
    SchMSwitch(EcuMode,ECUM_STATE_APP_POST_RUN)
  }
}

rule GoBackToRun initial false on TRIGGER {
  if (EcuMCompatibilityMode == ECUMCOMPATIBILITY_Run && EcuMode ==
    ECUM_STATE_APP_POST_RUN) {
    SchMSwitch(EcuMode,ECUM_STATE_APP_RUN)
  }
}

rule PrepShutdown initial false on TRIGGER {
  if (ComM_Mode_Channel1 == COMM_NO_COM_REQUEST_PENDING && EcuMode ==
    ECUM_STATE_APP_POST_RUN) {
    custom "EcuMCompatibility_OnPrepShutdown()"
    custom "Dem_Shutdown(null)"
    if (EcuMCompatibilityMode == ECUMCOMPATIBILITY_Sleep) {
      SchMSwitch(EcuMode,ECUM_STATE_GoSleep)
    }
```

```
    else {
      SchMSwitch(EcuMode,ECUM_STATE_GoOff1)
    }
  }
}

rule GoSleep initial false on TRIGGER {
  if (ComM_Mode_Channel1 == COMM_NO_COM_REQUEST_PENDING && EcuMode ==
    ECUM_STATE_GoSleep) {
    custom "EcuMCompatibility_OnGoSleep()"
    SchMSwitch(EcuMode,ECUM_STATE_SleepWaitForNvMWriteAll)
    custom "NvM_WriteAll()"
  }
}

rule GoOff initial false on TRIGGER {
  if (ComM_Mode_Channel1 == COMM_NO_COM_REQUEST_PENDING && EcuMode ==
    ECUM_STATE_GoOff1) {
    custom "EcuMCompatibility_OnGoOffOne()"
    custom "Rte_stop(null)"
    custom "ComM_DeInit(null)"
    SchMSwitch(EcuMode,ECUM_STATE_GoOff2)
    custom "NvM_WriteAll()"
  }
}

rule GoSleepNvMWriteAllFinished initial false on TRIGGER {
  if (NvMWriteAllJobMode == NVM_JOB_FINISHED && EcuMode ==
    ECUM_STATE_SleepWaitForNvMWriteAll) {
    custom "EcuM_GoHalt()"
  }
}

rule GoOff2 initial false on TRIGGER {
  if (NvMWriteAllJobMode == NVM_JOB_FINISHED && EcuMode ==
    ECUM_STATE_GoOff2) {
    custom "EcuMCompatibility_OnGoOffTwo()"
    custom "EcuM_GoDown()"
  }
}
```

4.4 Wakeup

The functionality for correct wakeup from sleep mode has to be fully configured in the BswM. But as it does not need any adjustments for backward compatibility, there are no modifications to be done.

5 Acronyms and abbreviations

5.1 Technical Terms

All technical terms used throughout this document – except the ones listed here – can be found in the official AUTOSAR glossary [6] or the Software Component Template Specification [2].

Term	Description
mode	A Mode is a certain set of states of the various state machines (not only of the ECU State Manager) that are running in the vehicle and are relevant to a particular entity, an application or the whole vehicle
state	States are internal to their respective BSW component and thus not visible to the application. So they are only used by the BSW's internal state machine. The States inside the ECU State Manager build the phases and therefore handle the modes.
phase	A logical or temporal assembly of ECU Manager's actions and events, e.g. STARTUP, UP, SHUTDOWN, SLEEP, etc. Phases can consist of Sub-Phases which are often called Sequences if they above all exist to group sequences of executed actions into logical units. Phases in this context are not the phases of the AUTOSAR Methodology.
mode switch port	The port for receiving (or sending) a mode switch notification. For this purpose, a *mode switch port* is typed by a `ModeSwitchInterface`.
mode user	An *AUTOSAR SW-C* or *AUTOSAR Basic Software Module* that depends on modes by *ModeDisablingDependency*, *SwcModeSwitchEvent*, *BswModeSwitchEvent*, or simply by reading the current state of a mode is called a `mode user`. A mode user is defined by having a require `mode switch port` or a *requiredModeGroup ModeDeclarationGroupPrototype*. See also section 2.
mode manager	Entering and leaving modes is initiated by a *mode manager*. A *mode manager* is defined by having a provide `mode switch port` or a *providedModeGroup ModeDeclarationGroupPrototype*. A *mode manager* might be either an `application mode manager` or a *Basic Software Module* that provides a service including mode switches, like the ECU State Manager. See also section 2.2.
application mode manager	An *application mode manager* is an *AUTOSAR software-component* that provides the service of switching modes. The modes of an `application mode manager` do not have to be standardized.

mode switch notification	The communication of a mode switch from the `mode manager` to the `mode user` using either the `ModeSwitchInterface` or *providedModeGroup* and *requiredModeGroup* *ModeDeclarationGroupPrototype*s is called *mode switch notification*.
mode machine instance	The instances of mode machines or *ModeDeclarationGroup*s are defined by the *ModeDeclarationGroupPrototype*s of the `mode managers`. Since a mode switch is not executed instantaneously, The RTE or *Basic Software Scheduler* has to maintain it's own states. For each `mode manager`'s ModeDeclarationGroupPrototype, RTE or *Basic Software Scheduler* has one state machine. This state machine is called *mode machine instance*. For all `mode users` of the same `mode manager`'s *ModeDeclarationGroupPrototype*, RTE and *Basic Software Scheduler* uses the same *mode machine instance*. See also section 2.2.
common mode machine instance	A 'common mode machine instance' is a special 'mode machine instance' shared by BSW Modules and SW-Cs: The RTE Generator creates only one `mode machine instance` if a *ModeDeclarationGroupPrototype* instantiated in a port of a software-component is synchronized (*synchronizedModeGroup* of a *SwcBswMapping*) with a *providedModeGroup ModeDeclarationGroupPrototype* of a Basic Software Module instance. The related `mode machine instance` is called `common mode machine instance`.
ModeDisablingDependency	An *RTEEvent* and *BswEvent* that starts a *Runnable Entity* respectively a *Basic Software Schedulable Entity* can contain a *disabledInMode* association which *references* a *ModeDeclaration*. This association is called *ModeDisablingDependency* in this document.
mode disabling dependent ExecutableEntity	A mode disabling dependent *Runnable Entity* or a *Basic Software Schedulable Entity* is triggered by an `RTEEvent` respectively a `BswEvent` with a `ModeDisablingDependency`. RTE and *Basic Software Scheduler* prevent the start of those *Runnable Entity* or *Basic Software Schedulable Entity* by the `RTEEvent` / `BswEvent`, when the corresponding `mode disabling` is active. See also section 2.2.
mode disabling	When a 'mode disabling' is active, RTE and *Basic Software Scheduler* disables the start of `mode disabling dependent ExecutableEntitys`. The 'mode disabling' is active during the mode that is referenced in the mode disabling dependency and during the transitions that enter and leave this mode. See also section 2.2.

OnEntry ExecutableEntity	A *Runnable Entity* or a *Basic Software Schedulable Entity* that is triggered by a *SwcModeSwitchEvent* respectively a *BswModeSwitchEvent* with *ModeActivationKind* 'entry' is triggered on entering the mode. It is called *OnEntry ExecutableEntity*. See also section 2.2.
OnExit ExecutableEntity	A *Runnable Entity* or a *Basic Software Schedulable Entity* that is triggered by a *SwcModeSwitchEvent* respectively a *BswModeSwitchEvent* with *ModeActivationKind* 'exit' is triggered on exiting the mode. It is called *OnExit ExecutableEntity*. See also section 2.2.
OnTransition ExecutableEntity	A *Runnable Entity* or a *Basic Software Schedulable Entity* that is triggered by a *SwcModeSwitchEvent* respectively a *BswModeSwitchEvent* with *ModeActivationKind* 'transition' is triggered on a transition between the two specified modes. It is called *OnTransition ExecutableEntity*. See also section 2.2.
mode switch acknowledge ExecutableEntity	A *Runnable Entity* or a *Basic Software Schedulable Entity* that is triggered by a *SwcModeSwitchedAckEvent* respectively a *BswModeSwitchedAckEvent* connected to the `mode manager`'s *ModeDeclarationGroupPrototype*. It is called *mode switch acknowledge ExecutableEntity*. See also section 2.2.
server runnable	A server that is triggered by an *OperationInvokedEvent*. It has a mixed behavior between a runnable and a function call. In certain situations, RTE can implement the client server communication as a simple function call.
runnable activation	The activation of a runnable is linked to the `RTEEvent` that leads to the execution of the runnable. It is defined as the incident that is referred to by the `RTEEvent`. E.g., for a timing event, the corresponding runnable is activated, when the timer expires, and for a data received event, the runnable is activated when the data is received by the RTE.
Basic Software Schedulable Entity activation	The activation of a *Basic Software Schedulable Entity* is defined as the activation of the task that contains the *Basic Software Schedulable Entity* and eventually includes setting a flag that tells the glue code in the task which *Basic Software Schedulable Entity* is to be executed.
runnable start	A runnable is started by the calling the C-function that implements the runnable from within a started task.
Basic Software Schedulable Entity start	A *Basic Software Schedulable Entity* is started by the calling the C-function that implements the *Basic Software Schedulable Entity* from within a started task.

Trigger Source	A *Trigger Source* administrate the particular *Trigger* and informs the RTE or *Basic Software Scheduler* if the *Trigger* is raised. A *Trigger Source* has dedicated provide `trigger port`(s) or / and *releasedTrigger Trigger*(s) to communicate to the `Trigger Sink`(s).
Trigger Sink	A *Trigger Sink* relies on the activation of *Runnable Entities* or *Basic Software Schedulable Entities* if a particular *Trigger* is raised. A *Trigger Sink* has a dedicated require `trigger port`(s) or / and *requiredTrigger Trigger*(s) to communicate to the `Trigger Source`(s).
trigger port	A `PortPrototype` which is typed by an `TriggerInterface`
triggered ExecutableEntity	A *Runnable Entity* or a *Basic Software Schedulable Entity* that is triggered at least by one `ExternalTriggerOccurredEvent` / `BswExternalTriggerOccurredEvent` or `InternalTriggerOccurredEvent` / `BswInternalTriggerOccurredEvent`. In particular cases, the Trigger Event Communication or the *Inter Runnable Triggering* is implemented by RTE or *Basic Software Scheduler* as a direct function call of the *triggered ExecutableEntity* by the triggering *ExecutableEntity*.
triggered runnable	A *Runnable Entity* that is triggered at least by one `ExternalTriggerOccurredEvent` or `InternalTriggerOccurredEvent`. In particular cases, the Trigger Event Communication or the *Inter Runnable Triggering* is implemented by RTE as a direct function call of the *triggered runnable* by the triggering runnable.
triggered Basic Software Schedulable Entity	A *Basic Software Schedulable Entity* that is triggered at least by one `BswExternalTriggerOccurredEvent` or `BswInternalTriggerOccurredEvent`. In particular cases, the *Trigger Event Communication* or the *Inter Basic Software Schedulable Entity Triggering* is implemented by *Basic Software Scheduler* as a direct function call of the *triggered ExecutableEntity* by the triggering *ExecutableEntity*.
execution-instance	An execution-instance of a `ExecutableEntity` is one instance or call context of an `ExecutableEntity` with respect to concurrent execution.
inter-ECU communication	The communication between ECUs, typically using COM is called `inter-ECU` communication in this document.

inter-partition communication	The communication within one ECU but between different partitions, represented by different OS applications, is called `inter-partition` communication in this document. It typically involves the use of OS mechanisms like IOC or trusted function calls. The partitions can be located on different cores or use different memory sections of the ECU.
intra-partition communication	The communication within one partition of one ECU is called `intra-partition` communication. In this case, RTE can make use of internal buffers and queues for communication.
intra-ECU communication	The communication within one ECU is called `intra-ECU` communication in this document. It is a super set of `inter-partition` communication and `intra-partition` communication.