



## Turing 1.3

### Table of Contents

<b>1</b>	<b>PURPOSE .....</b>	<b>4</b>
<b>2</b>	<b>SCOPE .....</b>	<b>4</b>
<b>3</b>	<b>ORGANIZATION OF THE WORK PRODUCTS.....</b>	<b>5</b>
3.1	OVERALL.....	5
3.2	UML USAGE.....	5
3.3	THIS DOCUMENT .....	6
<b>4</b>	<b>ARCHITECTURE BACKGROUND.....</b>	<b>6</b>
<b>5</b>	<b>PREREQUISITES .....</b>	<b>9</b>
5.1	GLOBAL CONSTRAINTS .....	9
5.2	ASSUMPTIONS .....	9
5.3	GLOBAL DECISIONS.....	9
5.4	FUNCTIONAL DECISIONS .....	13
5.4.1	Component identification.....	14
5.5	SOFTWARE ARCHITECTURE PRINCIPLES .....	14
5.5.1	Generic principles.....	14
5.5.2	Clarification of use-cases.....	17
5.5.3	Dependencies .....	18
5.6	REUSE STRATEGY .....	19
<b>6</b>	<b>STATIC VIEW.....</b>	<b>20</b>
6.1	META-MODEL.....	21
6.1.1	BSW – Basic Software.....	22
6.1.2	BSWExt – Basic Software Extensions .....	22
6.1.3	LIB - Libraries for shared use.....	22
6.1.4	SWC - Application Software Components.....	24
6.1.5	CDD - Complex Device Drivers .....	30
6.1.6	RES - Resource storage.....	31
6.1.7	RTE - Runtime environment.....	33
6.2	NAMING CONVENTION .....	35
6.3	GENERAL DIAGRAM .....	36
6.3.1	TA and TB use-cases.....	39
6.4	SYSTEM INTERFACES CONCEPT .....	39
6.4.1	System vs Functional interfaces .....	39
6.5	FILE STRUCTURE .....	42
6.5.1	SWC file structure .....	42
6.5.2	CDD file structure.....	45
6.6	TYPES.....	46
6.6.1	Standard types usage .....	46
6.6.2	User-defined types .....	50
<b>7</b>	<b>DYNAMIC VIEW .....</b>	<b>51</b>

7.1	SYSTEM LIFE-CYCLE .....	51
7.1.1	TA .....	51
7.1.2	TB .....	57
7.2	COMPONENT LIFE-CYCLE .....	59
7.3	OS USAGE.....	61
7.3.1	Scheduling .....	61
7.3.2	Using OS tasks as containers with sub-tasks, activated by prescalers.....	65
7.3.3	Demo SW scheduling .....	67
7.3.4	Rationale for FotaCtrl.....	68
<b>8</b>	<b>FUNCTIONAL BREAKDOWN .....</b>	<b>69</b>
8.1	COM INFRASTRUCTURE .....	69
8.1.1	Requirements .....	69
8.2	WARNINGS MANAGEMENT .....	71
8.3	WATCHDOG MANAGEMENT .....	72
8.3.1	TA.....	72
8.3.2	TB.....	73
8.3.3	WdgM integration.....	74
8.3.4	WdgM configuration.....	74
8.4	NVM MANAGEMENT.....	74
8.4.1	TA .....	74
8.4.2	TB.....	78
8.5	DIAGNOSTICS.....	78
8.5.1	TA.....	78
8.5.2	TB.....	78
8.6	DTCs MANAGEMENT.....	79
8.6.1	TA.....	79
8.6.2	TB.....	79
8.7	SAFETY .....	79
8.7.1	General design principles.....	80
8.7.2	End-to-End communication (E2E).....	81
8.7.3	Partitioning.....	83
8.7.4	Graphics subsystem safety .....	86
8.7.5	Peripheral protection unit (PPU) .....	88
8.7.6	Other safety mechanisms .....	88
8.7.7	Reaction on safety failure .....	89
8.7.8	Example design.....	89
8.8	I/O HARDWARE ABSTRACTION.....	90
8.8.1	IoHwAb.....	90
8.9	TIMING SERVICES.....	93
8.9.1	TmExt.....	93
8.10	GUI SUB-SYSTEMS .....	94
8.11	SYSTEM MODE MANAGEMENT .....	94
8.11.1	The problem .....	94
8.11.2	Analysis of the needs .....	94
8.11.3	Requirements & constraints .....	95
8.11.4	Definition of mode management.....	96
8.11.5	Alternatives .....	96
8.11.6	Decisions .....	96
8.11.7	MVC Solution .....	96
8.12	SYNCHRONIZED ACTIVATION.....	97
8.13	DEBUG AND TRACE .....	98
8.13.1	DLT .....	98
8.13.2	DET .....	105
8.13.3	RTM.....	105
8.14	DISPLAY MANAGEMENT .....	109

8.14.1	Design decision.....	109
8.15	POWER ON/OFF SEQUENCE .....	110
8.15.1	Design decision.....	111
8.16	BATTERY MANAGEMENT.....	112
8.16.1	Battery state .....	112
8.16.2	Actions in case of degraded mode .....	113
8.16.3	Overall architecture proposal .....	113
<b>9</b>	<b>BUILD .....</b>	<b>114</b>
9.1	GLOBAL DEFINES .....	114
9.2	PRODUCT VARIANT MANAGEMENT WITH CONDITIONAL COMPILATION.....	114
9.3	COMPONENT LEVEL DEFINES .....	116
9.4	COMPILER AND LINKER OPTIONS .....	117
<b>10</b>	<b>TESTING .....</b>	<b>117</b>
<b>11</b>	<b>TOOLS.....</b>	<b>117</b>
<b>12</b>	<b>APPENDIX.....</b>	<b>118</b>
12.1	TRAININGS .....	118
12.1.1	Agenda for Design Champions to explain individual components.....	118
12.1.2	Check-list for detailed design completeness .....	119

**Revision History**

\* Version is increased on every release.

Version	Date	Author	Description of changes
Turing 1.2 (hidden)			
Turing 1.3 (hidden)			
Turing 1.4 (hidden)			
Turing 1.5			
22	26-Jan-2016	B. Shumlev	Update of components kind chapter with recommendation for data-pool connector implementation.
23	25-Feb-2016	B. Shumlev	Added DET usage chapter
24	29-Mar-2016	B. Shumlev	Added RTM guideline chapter
25	25-Jul-2016	V.Gorchev	Battery state monitoring explained

**1 Purpose**

This document describes Driver Information Software Architecture, which is intended to be used in combination with several HW platforms.

It is based on Lessons Learned of Visteon Classic and xJCI Driver Information platforms and products as well as OEM needs that are foreseen for the near future.

The main sources of information and work products, identified so far are:

- Visteon Classic → SOSA bookshelf + initial Software Platform for Autosar.
- xJCI → STK, Gen2, A2TOM
- OEM projects supplying use-cases: Ford, VW, BMW, DAG, Renault
- AUTOSAR specification

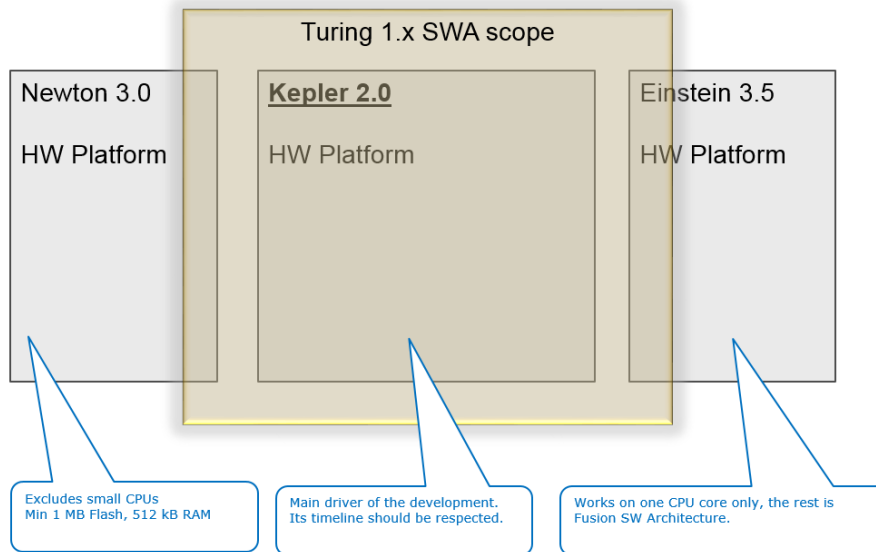
The target audience consists of all roles involved in design and development of reusable software architectures, SCRUM teams executing the platform and all people, developing projects, based on it.

It has being decided by management to have distinguishable names for SW architectures because:

- Confusion happens when we speak about HW platforms and SW architectures with the same name.
- One SW architecture is deployed to several HW platforms.

The presented software architecture is called **Turing**, named to **Alan Turing** → [http://en.wikipedia.org/wiki/Alan\\_Turing](http://en.wikipedia.org/wiki/Alan_Turing)

**2 Scope**



Target business domains:

- Included: Mid-end and high-end DI Clusters, HUDs, **Body Controllers**
- Excluded: **DI Displays**, Infotainment, Central stacks, Audio

### 3 Organization of the work products

#### 3.1 Overall

Software Architecture is not one single document, it is a set of interconnected and released together documents as follows:

- This document, which contains all major assumptions, decisions and general choices.
- UML model with details about concepts, global solutions and solutions per sub-systems.
- Individual design documents per frameworks & sub-systems, including design aspects and guidelines for them.
- UML model for instance of the architecture, representing concrete Demo Application design.
- Supporting documentation about tool-chain usage.

Note: Detailed design of individual components is not part of Software Architecture, but part of Bookshelf.

#### 3.2 UML usage

In the UML Model there are several different parts:

- **Meta-model**, which describes all SWA elements and they relations.
- **SWA High-Level Design**, which respects the meta-model constraints.
- **Bookshelf Design**, which describes the details of the individual assets.
- **SWA Reference Design**, based on High-Level Design and represents SWA of particular reference applications.

Copies of this document are uncontrolled

Page 5 of 119

The used UML tool is Rhapsody 7.6.1 with migration to 8.1.1 later on.  
A2TOM 3.1 profile is used for the beginning and 4.x profile will be applied when ready.

### 3.3 This document

It explains:

- Global assumptions that does not depend of SWA team
- Global decisions based on global assumptions and features that platform needs to cover.
- Global design rules for software construction.
- Concepts and frameworks.
- Static and dynamic view of the SWA at global level.
- Relations to the tool-chain (so-called "ecosystem").

The major points have individual chapter for explanation.

## 4 Architecture Background

Software Architecture for AUTOSAR and Non-AUTOSAR will cover the automotive ECUs supporting driver information, multi-function display, and center stack (TBC) electronics functionality. It will provide a software platform on which to build mid to mid-high level driver information software. SWA is initially targeted for the Kepler 2 HW platform.

The goals of SWA are to develop an architecture to improve upon the deficiencies of the Driver Information Architectures of xJCI and Visteon Classic (DI-Gen2 and DI-SOSA) and to achieve the quality attributes necessary to support the software development lifecycle.

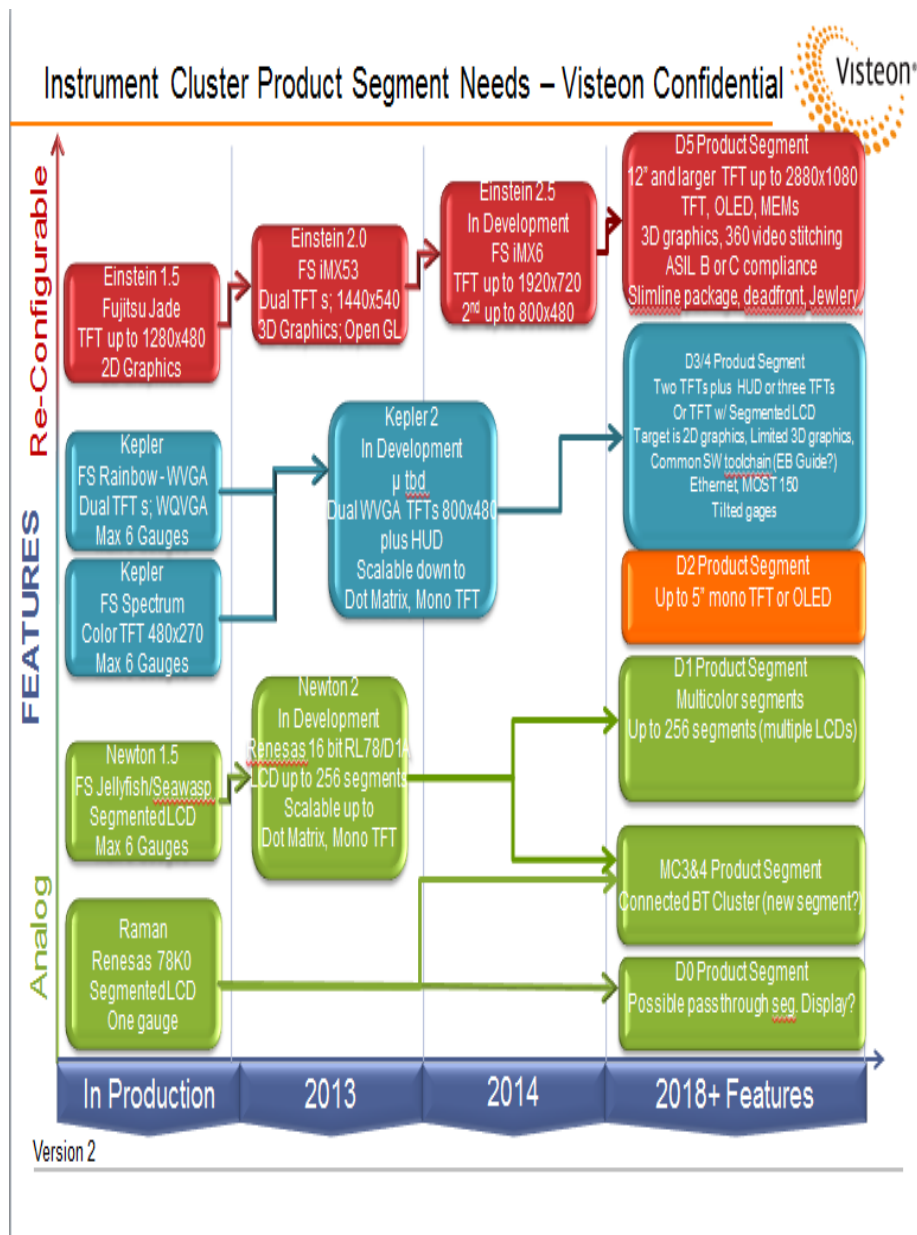


Figure 1: Product platform roadmap.

## Kepler II/ Micro/Cluster Scalability

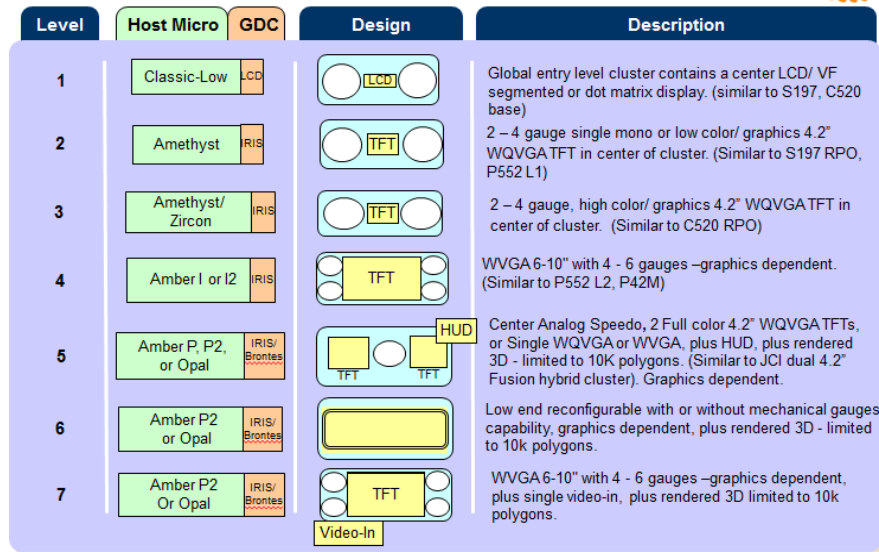


Figure 2: Product platform scalability for Kepler 2.

DI-SOSA Architecture Deficiencies	
Quality Attribute	Explanation of Deficiency
Deployable	poor due to poor maintainability and testability
Interoperability	Not considered
Maintainability	poor product implementation
Monitoring	Not supported
Portability	Insufficient abstraction; architecture supported but implementation did not
Scalability	Reduced by tight coupling
Security	Not considered
Testability	Had only limited support

Table 1: DI-SOSA deficiencies with an explanation of the individual quality attribute deficiency.

DI-SOSA Architecture Positive Attributes	
Quality Attribute	Explanation of Attribute
Performance	well supported; designed specifically for this
Reliability	well supported
Usability	well supported

Table 2: DI-SOSA positive attributes with an explanation of the attribute.



## 5 Prerequisites

### 5.1 Global constraints

This chapter describes non-negotiable constraints, provided by management at the phase of definition of next generation platforms. Any change in these might lead to significant impact to the decisions made below, i.e. may invalidate all work products in the derived projects.

Constraint	Description
[C1]	[C1.1] Software Architecture has to cover the needs of Driver Information projects that are Autosar based. [C1.2] The selected distribution package is Vector MICROSAR 4.2.1 This version is called TA.
[C2]	For low-cost markets Software Architecture has to support Non-Autosar variant, shortly called TB. The same Software Components and Complex Device Drivers shall be reused from TA.
[C3]	Maximal reuse of components, tools and methodologies between TA and TB is required.
[C4]	Lesson learned from Kepler 1, STK, and Gen2 to be taken into account.
[C5]	The platform Einstein 3.5 which will use multi-core CPU will have to run Turing 1.0 software system on its driver information core/ VIP processor.
[C6]	The platform will implement a configuration and generation framework.

### 5.2 Assumptions

These are points that might never be answered by anybody or cause a lot of discussions at different levels or simply it is not correct to be decided by SW architects. Nevertheless, they must be fixed in order to have stable basis for design.

As far as constraints above are not changed, the modifications of the points below will affect the scope of the Software Architecture.

Assumption	Description
[A1]	It is not requested to have backward compatibility with any former architecture, nor to previous versions of Autosar.
[A2]	Requirements collection/approval for Turing will late. System engineers will be assigned later on. It is responsibility of the SW Architects to define the features of the platform, based on information, collected from first projects expecting Turing 1.0 and predicted needs of the targeted market segment. It is expected to have significant impact to decisions and solutions already made, when all requirements are ready.

### 5.3 Global decisions

Here are the main decisions, taken by the SW Architects. Every change in those drastically affects the subsequent decisions and design solutions.

Template for decisions:

D#	
Status	Draft   Proposed   Approved
Category	General, SWC, CDD, TA, TB, Safety, Tools, Configuration, OS
Description	
Rationale	
Comments	

Copies of this document are uncontrolled

Page 9 of 119

## Software Architecture Specification

D0	<b>Multicore</b>
Status	Approved
Category	General
Description	<ul style="list-style-type: none"> <li>TA will be placed on a single core, even when dual core is used. Second core might be used for graphics stack.</li> <li>TB will never be extended to work on several cores.</li> </ul>
Rationale	TA: Safety – freedom from interference to be ensured between automotive and GUI parts. TB: Not feasible to be designed and implemented within a reasonable time-frame and resources.
Comments	This decision allows safety and non-safety to be separated by HW. Turing 1.0 will not utilize the multicore AUTOSAR capability. For later phase, multicore will be investigated for Spansion Opal MCU.

D1	<b>Same Design Rules for TA and TB</b>
Status	Approved
Category	General
Description	<p><i>TA and TB must use the same rules for high-level design and construction. They are treated as parts of the same SW Architecture, not as independent things working together.</i></p> <p><i>TA is the basic one for SWA. Components/assets will be developed to match it → no wrappers; pure &amp; clean design &amp; implementation, If needed TB variant will create wrappers/adapters and/or will re-create some parts to ensure the reusability (Ex. no more usage of remapping via RI files).</i></p>
Rationale	[C3],[C4] It is needed to have possibility to share and exchange work products between them. They must be able to be upgraded together with minimal effort.
Comments	

D2	<b>TB scope is subset of TA scope</b>
Status	Approved
Category	General
Decision	<p><i>TA and TB should be kept as close as possible in terms of features and services/tools they provide.</i></p> <p><i>TA will have the bigger set. TB will have a sub-set. What will be the omitted part is decided by SW architects, after evaluation of the feasibility, risks and effort of all features put in the scope.</i></p>
Rationale	The wish of the users of the platform is to have the same functionality for TA and TB. Even rough evaluation shows that this is impossible within the given timeline, because of complexity of TA 4.2.1. Also it does not make sense to recreate and revalidate something that is already purchased. Even if we do it we will have footprint which does not fit with the smaller CPUs that will be used for TB.
Comments	TB will not support unique features from TA. Unique TB features needs shall be addressed by project teams, not by the SWA and platform team.

D3	<b>OSEK used in TA and TB</b>
Status	Approved
Category	OS
Decision	<p><i>TA and TB should use the same OS. We suggest OSEK. In Vector case this is MicroSAR OS. If supplier is changed we expect OSEK specification to be covered. For the beginning the selected scalability class is SC1, with the intention of migrating to SC3 later on.</i></p> <p><i>Backup solution: TOS w/o extended tasks for low market segment. Guideline to be made for TOS.</i></p>
Rationale	Needs of the projects cannot be satisfied with TOS neither with DI Kernel. We cannot make OS by ourselves. We need to purchase. AUTOSAR OS will be purchased, so it will

## Software Architecture Specification

	be easy to extend the usage to more projects. We will know/test already all of it for TA, thus no retesting needed for TB.
Comments	
D4	<b>Task scheduling</b>
Status	Approved
Category	OS
Decision	<p><i>Rate-monotonic schema → 70 % max overall CPU to be schedulable. Split of the system CPU resources will be done with respective quotas:</i></p> <ul style="list-style-type: none"> <li><i>Automotive part: tasks are in non-preemption groups – max 70 % from the max overall; if preemption needed, it must be explicitly approved by Design Champion and must be done via RTE with the needed protections.</i></li> <li><i>Graphical part: up to 5 (preferable up to 3) preemptive tasks – max 30 % from the max overall; GPU utilization needs guidelines to be created → separate topic.</i></li> </ul>
Rationale	Lessons learnt from xJCI projects.
Comments	More than 5 preemptive tasks will be problematic. There is a case-study for that topic.
D5	<b>Input work products</b>
Status	Approved
Category	Porting
Description	<p><i>The next aspects of reuse from former work products is decided:</i></p> <ul style="list-style-type: none"> <li><i>SOSA services to SWCs, CDDs, LIBs</i></li> <li><i>STK, Gen2 components to SWCs, CDDs, LIBs</i></li> </ul> <p><i>Investigation DI Kernel was made. The decision is not handover it.</i></p>
Rationale	Reuse of existing bookshelf components, tools and methodologies. Anything that is obsolete or ineffective won't be transferred. But good parts will be. DI Kernel scheduler does not provide the needed RTOS support.
Comments	<p>See the chapter for bookshelf below.</p> <p>Consequences of [A1] and this decision:</p> <ul style="list-style-type: none"> <li>Architects won't try to keep backward compatibility. This means that any reuse/porting will be in one direction → towards Turing 1.0. I.e. Turing 1.0 components cannot be used in Kepler 1/SOSA projects.</li> <li>The point-to-point communication links of SOSA need to be modified to support 3-point communication (via DP).</li> </ul>
D6	<b>Common diagnostics</b>
Status	Approved
Category	General
Description	<ol style="list-style-type: none"> <li><i>The decided diagnostic approach will be the same for TA and TB.</i></li> <li><i>The system will support OEM diagnostics, developer diagnostics/debugging, UI diagnostics/debugging.</i></li> <li><i>Measurement and calibration will rely upon XCP.</i></li> </ol>
Rationale	Reuse, risk reduction, speeding the development and testing. Ensure synergy with the way OEM is testing.
Comments	
D7	<b>Design paradigms</b>
Status	Approved
Category	General/Reuse
Description	<p><i>The next design paradigms and patterns are used as basis for creation of Software Architecture:</i></p> <ul style="list-style-type: none"> <li><i>Autosar BSW, CDD and SWC concepts.</i></li> <li><i>SWC internally will use Model-View-Controller pattern.</i></li> <li><i>Library pattern (LIB) will be maintained explicitly.</i></li> </ul>

## Software Architecture Specification

	<ul style="list-style-type: none"> <li><i>Specific pattern for resource storages (RES) must be maintained.</i></li> <li><i>Datapool (DP) will be utilized for component communication. Separate data pools will be used for HMI and automotive SWCs, when necessary.</i></li> </ul>
Rationale	Ensuring clarity, robustness and reusability of products. Extremely important for maintenance of multiple component- and product- variants.
Comments	Reused artefacts needs to be renamed not to conflict with TA ones.

<b>D8</b>	<b>Complex Device Drivers</b>
Status	Approved
Category	CDD
Description	There is only one set of CDDs for TA and TB. They will utilize TA or TB CDD MCAL, unless functionality not supported. They will interface with HW directly only when TA or TB CDD MCAL doesn't support. The same component system interface will manage SWCs and CDDs in TA and in TB variants.
Rationale	Reuse across variants. PC Simulation to be available for TA and TB.
Comments	

<b>D9</b>	<b>Libraries</b>
Status	Approved
Category	LIB
Description	Common libraries to be established in the new bookshelf. They must be used by internally developed components. Libraries are not allowed to call/depend on SWCs, CDDs and BSWs.
Rationale	Reuse. Not possible to share SWCs and CDDs between TA and TB if not standardized.
Comments	Rules how to solve conflicts with external libraries need to be defined. There is an impact to classic Visteon and xJCI components, as the libraries will be a migration of existing libraries and new libraries.

<b>D10</b>	<b>Standard types</b>
Status	Approved
Category	LIB/Types
Description	All developed components will use the Autosar BSW types → even above the RTE. This includes TB variant as well.
Rationale	Lessons learnt from MQB, Edison etc. This is the same as the decision as MFA2 project.
Comments	Rules how to solve conflicts with external types need to be defined.

<b>D11</b>	<b>Time services</b>
Status	Approved
Category	Services
Description	All developed TA and TB components will use the same set of time services.
Rationale	Reuse
Comments	Mapping to these can be done to provide the same time base to external components.

<b>D12</b>	<b>Datapool</b>
Status	Approved
Category	Datapool
Description	TA and TB will use the same concept for datapool. TB will have its own generator, designed and maintained by platform team. Datapool will implement TA 4.2.1 features that improve upon 3.1, but only the minimal needed set.
Rationale	Reuse.

	Feasibility of the implementation on our side. Footprint of TB.
Comments	
D13	<b>SW Components Control</b>
Status	Approved
Category	SWA
Description	SWCs and CDDs for TA and TB will have centralized control from outside. They must support life-cycle state-machine that is identical for all and is encapsulated in a reusable bookshelf asset.
Rationale	At run-time different set of components are active in different situations. Components have dependencies, so they need to be chained to operate properly. So, they need to be enabled/ disabled and/or started/stopped independently. In TA case this might be done with Autosar means, but they are hard to be resembled in TB variant, thus we need to have our own system. Experience from xJCI projects will be reused.
Comments	
D14	<b>Deployment Framework</b>
Status	Approved
Category	Tools
Description	Turing SWA will define mechanisms for configuration of individual components and for services for shared use. It will be combination of A2TOM tools and additional Java tools, working together.  It is important that source code regeneration of the whole system must be able to be done via command line. When configuration files are ready they must be able to be run at once to recreate all the parts of the source code, considering the selected variant of the product.  By "Java tools" we mean technologies that are based on Java or massively used with Java worldwide. Such as: artop, artext, Eclipse + plug-ins, Rhapsody + plug-ins, xtext, xtend, xml, xslt, json, etc.  Turing 1.0 definitely restricts these technologies to be part of the tool-chain that will be deployed → .Net, C#, VBA, VB, Perl, Python. In the beginning there will be some reused stuff using these. With the time they will be replaced.
Rationale	Regarding deployment of code and configuration, strong rules and unification of methodology and tools are required to reduce the learning curve for software team members and the proliferation of unique solutions.
Comments	"make" file will be required for each component. For production of configuration files, command line execution is mandatory, no matter whether GUI is there or not. Java is able to ensure both at the same time. Because Java must be started for each code generator, it is needed to have a way to launch it once for all components during the build.

#### 5.4 Functional decisions

There are requirements that cause major changes in the SWA, if they are not taken into account since the beginning. They are not for particular project, but a collection of such from several places. Usually stronger than needed. The goal is to be traced individually.

Different features are added incrementally to the demo applications. Priorities are given by the customer projects.

#### 5.4.1 Component identification

AUTOSAR component version support → own components should not have GetVersion info till synch with RTC is discovered. If it is required, we will start supporting it.

DLT identification → Unique within a build and for one given definition of the system tree.

It is not mandatory TA and TB to have the same identification.

### 5.5 Software Architecture Principles

#### 5.5.1 Generic principles

This chapter contains word-wide principles of software architecture creation as well as specific treatment of those. They are explicitly expressed as rules and explained here because they are the foundation of stability and reuse.

The information below is based on article "Design Principles and Design Patterns" Copyright (c) 2000 by Robert C. Martin.

P01	<b>Encapsulation is required at any level in the system</b>
Description	<p>This is another explanation of <b>Dependency Inversion Principle</b>:</p> <p><i>High-Level modules should not depend on Low-Level modules. Both should depend on abstractions.</i></p> <p><i>Abstractions should not depend on details. Details should depend on abstractions.</i></p> <p>Dependency Inversion Principle states that we should decouple high level modules from low level modules, introducing an abstraction layer between the high level classes and low level classes. Furthermore it inverts the dependency: instead of writing our abstractions based on details, we should write the details based on abstractions. In the classical way when a software module (class, framework ...) need some other module, it initializes and holds a direct reference to it. This will make the 2 modules tight coupled. In order to decouple them the first module will provide a hook (a property, parameter ...) and an external module controlling the dependencies will inject the reference to the second one.</p> <p>By applying the Dependency Inversion the modules can be easily changed by other modules just changing the dependency module.</p> <p>See the dedicated chapter for the dependencies.</p>
Exceptions	Allowed only with agreement with Design Champion/SW Architect.
Comments	<p>3<sup>rd</sup> party components and/or sub-systems that will be integrated are going to be wrapped up and isolated, in order not to mess others.</p> <p>Q: How does low layer to notify upper layer? A: Via observer pattern or callbacks that are configured by a tool or by hand, but the subject has no idea who is notified.</p>

P02	<b>Interface Segregation Principle</b>
Description	<p><i>Clients should not be forced to depend upon interfaces that they don't use.</i></p> <p>When interfaces are written we should take care to add only methods that should be there. Otherwise the classes/components implementing the interface will have to implement those methods as well.</p> <p>As a conclusion Interfaces containing methods that are not specific to it are called polluted or fat interfaces. We should avoid them.</p>
Exceptions	Not allowed for manually managed code.

Copies of this document are uncontrolled

Page 14 of 119

	Allowed for auto coding systems when they are proven to be safe. For example inclusion of header files that are not used in the code, but the code generator never creates references to the content of these header files. Normally, these inclusions are part of the code template and in some cases they remain unused.
Comments	For example Autosar RTE makes stubs for not used ends of Sender/Receiver ports. This is temporary state till all senders and receivers appear in the system or a permanent state for some variants of the product where these ends are not needed. In that case the overhead is known and controlled, thus RTE description remains the same for all variants.

P03	<b>Single Responsibility Principle</b>
	<i>A class should have only one reason to change.</i>
Description	<p>In this context a responsibility is considered to be one reason to change. This principle states that if we have 2 reasons to change a class, we have to split the functionality in two classes. Each class will handle only one responsibility and in the future if we need to make one change we are going to make it in the class which handles it. When we need to make a change in a class having more responsibilities the change might affect the other functionality.</p> <p>In other words: Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.</p> <p>In SWA:</p> <ul style="list-style-type: none"> <li>Any SWC and CDD will have dedicated responsibility for particular functionality. If it is complex it will be broken down in sub parts, each dedicated to one and only atomic feature. Also patterns will be used to strongly specify the role of each component/class (MVC, Classical patterns).</li> <li>Shared libraries will have single purpose too → grouping of the functions per domain like: math, filters, crypto etc. Internally every library will have blocks that will go always together and might be selectable at compile time.</li> </ul>
Exceptions	Not allowed.
Comments	Single Responsibility Principle was introduced by Tom DeMarco in his book Structured Analysis and Systems Specification, 1979. Robert Martin reinterpreted the concept and defined the responsibility as a reason to change.

P04	<b>Open Close Principle</b>
	<i>Software entities like classes, modules and functions should be <u>open for extension</u> but <u>closed for modifications</u>.</i>
Description	<p>It is a generic principle. Software entities like packages, libraries, classes, components and functions should be open for extension but closed for modifications. Once the entity is created then extension is applied and we preserve the previously done stuff. Some of the obvious benefits are backward compatibility and regression testing.</p> <p>This can be ensured by use of Abstract Classes and concrete classes for implementing their behavior. This will enforce having Concrete Classes extending Abstract Classes instead of changing them. Some particular cases of this are Template Pattern and Strategy Pattern.</p>

	<p><b>Nota bene:</b> If there is abstract base class or reusable functionality that is badly designed, before applying this principle we need to fix it.</p> <p><b>Liskov's Substitution Principle</b> is extension of that one.</p> <p><i>Derived types must be completely substitutable for their base types.</i></p> <p>In terms of behavior it means that we must make sure that new derived classes are extending the base classes without changing their behavior. The new derived classes should be able to replace the base classes without any change in the code.</p> <p>In SWA:</p> <ul style="list-style-type: none"> <li>• Reuse across product variants desperately needs this principle to be followed. We need to share same SWCs &amp; CDDs across TA and TB for example, but they need to share/inherit common functionality.</li> <li>• OOP/OOD will be used in some cases, when helpful for the factorization.</li> <li>• C++ also will be used in high-level variants.</li> </ul>
Exceptions	Allowed with agreement of Design Champion/SW Architect.
Comments	There might be temporary solutions/ patches or wrappers pf 3 <sup>rd</sup> party components that are not necessarily following this.
P05	<b>Reuse Principles</b>
Description	<p>These are 3 interconnected principles:</p> <p><i>Release Reuse Equivalency: The granule of reuse is the granule of release.</i></p> <p><i>Common Closure: Classes that change together, belong together.</i></p> <p><i>Common Reuse: Classes that aren't reused together should not be grouped together</i></p> <p>In SWA:</p> <ul style="list-style-type: none"> <li>• The granule of release is <i>Component</i>; <i>SWC</i> is only one kind of component.</li> <li>• MVC pattern for SWCs and CDDs with their parts are examples of Common Closure and Common Reuse. Also SWA frameworks.</li> <li>• All these principles are applied to individual components and to sub-systems/packages that can be reused as a whole → CAN Stack, LCD Segment system, GUI models + generated code, frameworks;</li> </ul>
Exceptions	Allowed with agreement of Design Champion/SW Architect.
Comments	Globally and finally they have to be respected, but temporary they can be broken to make available intermediate builds in favor of testing. These are usually broken in rapid prototyping or simulation environments. Target code with many variants needs to respect those at the end → otherwise will become not manageable.

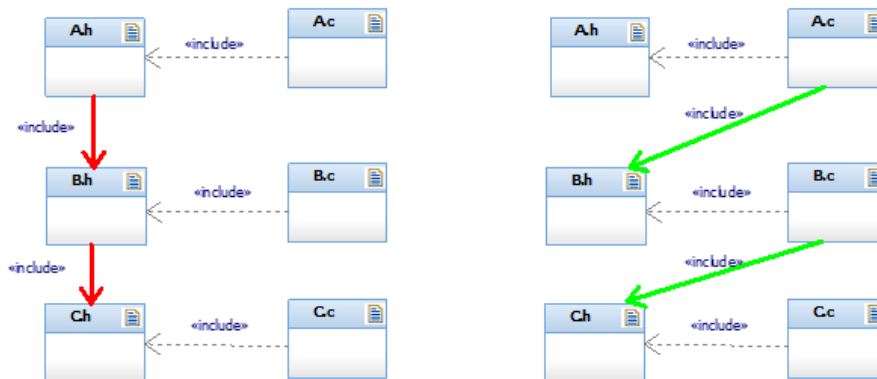


## 5.5.2 Clarification of use-cases

### 5.5.2.1 Avoid dependency promotion

This is the case when dependencies are like  $A \rightarrow B \rightarrow C$ , but we do not want to have  $A \rightarrow C$ . Here A, B and C are files, components, packages, sub systems. For simplicity we will explain with files.

The problem is when we have direct header inclusions, the solution is not to have such:



The red dependencies are not advised, so dependencies are reduced and preprocessing speeds up.

One particular use-case that lead people to include headers in headers and explanation of methods to avoid it → A.h needs type definitions declared in B.h.

Ask the question: Do the users of A.h need these definitions in B.h?

- YES → This is already public interface of A, so all public types must be in its header and not in other sub-modules or foreign modules. In that case B.c needs to include and to use them from A.h. Ask also the question: How many public headers you want for one module – 1, 2 ... 101 ?
- NO → obviously in that case they must not be propagated:
  - Either it exposes private parts outside module A, which is direct breaking of the encapsulation or
  - It propagates foreign module's definitions as they are its own, which is jump over one level in the architecture (whatever the direction is), which is definitely a bad practice → module must communicate only with its direct neighbors.

If the order of inclusions in \*.c file is done in the right way, then headers will see the needed definitions from the headers that are included before them. Keep the following order:

- Global header files with type definitions first
- Global reusable headers of libraries for shared use
- Public header file of the module
- Header files that are valid for all module level parts.
- Header file of the specific module part, which this \*.c/cpp file is dedicated to.

For sure, in the reality there are many cases where there is no control of the inclusions and this approach cannot be kept. The idea is that for self-made design/code there is real chance to minimize inclusions of headers in headers and to preserve the encapsulation.

Specific situation is when OOP is used in C language → some private stuff is published into the header file, i.e. exactly as in C++/ Java class declaration. But there it is protected. So, it must be protected in C case with #define macros for private/ protected visibility, which macro will not be

defined outside the module, thus these definitions will be removed by the preprocessor (See xJCI UML Autocode generation for example).

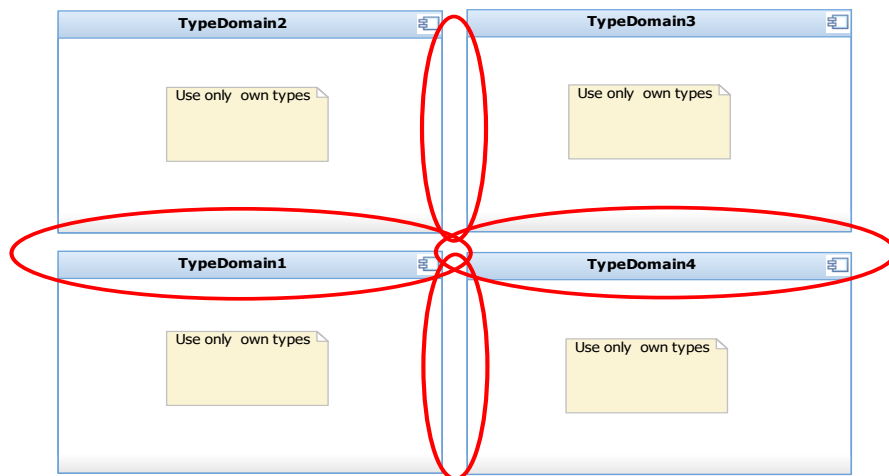
There are two cases where inclusion of header in header is really useful, if used with caution:

- By default the public header file is one, but split in two parts:
  - Frozen part that is designed once and for all.
  - Auto-generated part with parameters, included into the first one.
- One header file groups only other header files in a specific order and does not define any own things. This is to deploy the group at several places in the same way and to be able to replace at once this group with another one, which defines the same things.

### 5.5.2.2 Type segregation

This explanation is for domain specific types, not for global types that are meant to be used everywhere in any domain.

In hybrid projects there are many sets of type definitions. They must be used only within the respective domain and not outside.



Type conversions should be used only at the borders between the domains. Never deploy foreign types inside the domain. This directly breaks the encapsulation principle.

Note that C and C++ are also different domains. So, if one components uses both it has to segregate the type usage too.

It is important every TypeDomain to export to outside minimum public type definitions.

In case of having horizontal or vertical layering, TypeDomains are allowed to be converted only from/to the closest direct neighbor TypeDomain. In the picture above TypeDomain1 and TypeDomain3 even are not treated as neighbors. This is essential for decoupling.

### 5.5.3 Dependencies

Some rules to be respected:

- When there is dependency between SWA structural elements, it is based only on their public interfaces. Never make dependency to private interfaces nor to private content.

Copies of this document are uncontrolled

- Outer elements can access only the direct inner elements in the next nested level. I.e. jumping over layers in any direction is strongly forbidden.
- Elements at the same level (so-called siblings) can depend on each other if this does not violate rules, provided for the sub-system they belong to (TA SWCs are siblings that must not interact directly with each other for example).
- Inner elements must not have any knowledge about the outer elements. Callbacks needs to be abstracted.
- Libraries for shared use can be called from all elements.
- Libraries cannot depend on non-library elements.
- Libraries can call each other as well.
- The dependencies between elements must not form cycles.

When dependency between two elements/components is needed it is preferable to be indirect via mediator, for example: Observer pattern, via RTE/datapool interfaces, remapped required interface files. This is necessary to ensure the possibility to make several product variants out of the same encapsulated elements.

In general TA and TB variants Autosar restrictions are taking place as a basis. Then the following addendums are put on top:



- LIBs can be called from SWCs and CDDs directly. LIBs always go together with the components that are reused.

Try to avoid dependencies between CDDs in order to be able to reuse them separately. The idea is to have vertical reusable stacks, that might depend only on the standard BSW::MCAL interfaces (i.e. TA and TB to provide the same MCAL interface)

## 5.6 Reuse strategy

Based on the design decisions above reuse strategy is divided in categories. Each one represents different aspect:

Reuse Category	Description
Shared assets for global reuse	<p>Because there are components and sub-systems that are by default dedicated for global shared reuse across many projects (even outside SWA), but they are put in the scope of SWA, and there is no separate team do design/ construct/ update them, they will be put in the <i>bookshelf</i>.</p> <p>The currently identified ones are:</p> <ul style="list-style-type: none"> <li>• CDD for Stepper Motors</li> <li>• CDD for Inter-processor Communication Stack (IPC)</li> <li>• LIBs → Libraries for shared reuse (see next chapters)</li> </ul> <p>More to be defined on-the-fly.</p>
Scale down BSW	<p>For TA and for TB separately → guideline will be provided how projects can safely remove parts of them, when they do not need some features.</p> <p>E.g. from TA: BSW will support E2E and will not be mandatory for all customers, DLT can be scaled down depending upon desire monitor-ability.</p>
Same assets used in TA and TB	<p>SWCs, CDDs, DPs, LIBs(same as above), OS, RES</p> <p>E.g.: Provide rules on OSEK configuration, scheduling mechanism for non-UI vs UI, CPU load measurement system, ecosystem (all tools to be used)</p>
Reuse across different MCUs	<p>How to construct CDDs, how to deploy them to different MCUs. CDDs are designed to use the same SRV and HWAL parts combined with various MCALs.</p>
MVC + DP reuse at project's side	<p>Guideline for user projects about how to create as much as possible reusable SWCs, based on their specification. Demo application will be based on this.</p>

	<p>Training materials with examples to be prepared.</p>  <p>DI-MVC.pptx</p>
Reuse of the design	<p>High-level design and detailed-level design are described in UML and interlinked documents. Projects, derived from this SWA are not starting their design from scratch. They take the prepared parts (packages) from the bookshelf and reuse/extend them on demand.</p>
Parametric components/assets	<p>How to create variants of components, based on:</p> <ul style="list-style-type: none"> <li>• Manual parameterization</li> <li>• Code generation from databases (XML, XLS, etc.)</li> <li>• Functionality</li> </ul>
Single and bi-processor/ multicore cases.	<p>Rules and guidelines about the distribution of SW system, which maximize the reuse of same components in the next cases at the same time:</p> <ul style="list-style-type: none"> <li>• The whole system on one core, no MMU</li> <li>• The whole system on one core, with MMU → safety and non-safety partitions</li> <li>• The system in dual-core (Opal case) → what to be where &amp; inter-core communication;</li> <li>• The system on two separate CPUs → IPC connection to Einstein 3.5</li> </ul> <p>Map for applicability for TA and TB variants will be elaborated.</p>
Framework usage	<p>Two main ways of reuse are possible:</p> <ul style="list-style-type: none"> <li>• Use all provided SWA features to develop own components.</li> <li>• Use 3<sup>rd</sup> party components – in source and/or object form &amp; wrappers only to plug them into the system.</li> </ul> <p>I.e. hybrid systems are possible.</p>
System assembly	<p>The following file explains several cases of reuse when assembling product variants:</p>  <p>Microsoft Excel Worksheet</p>
Reuse outside this SWA	<p>Guidelines. Examples:</p> <ul style="list-style-type: none"> <li>• How Steppers, IPC, LIBs can be reused in Newton3</li> <li>• How to port GUI HMI systems by removing Autosar dependency.</li> </ul>
Conceptual reuse	<p>Documentation and demo application with design principles for concepts that might be re-implemented in completely different software architectures:</p> <ul style="list-style-type: none"> <li>• Multi GUI sub-systems support concept</li> <li>• Warning management concept</li> <li>• Resource management concept</li> <li>• Time services</li> </ul>

## 6 Static View

This chapter explains SWA from static perspective.

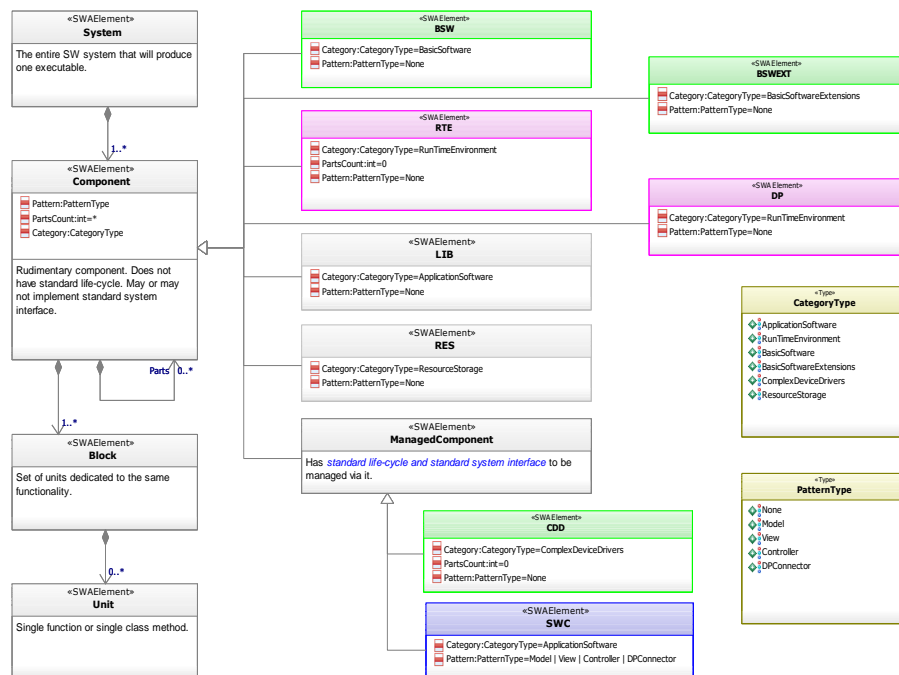
## 6.1 Meta-model

Meta-models is set of class diagrams, explaining what the approved elements are and which relations between them are allowed. The consequences are:

- If element/relation is not described in the meta-model, it does not exist and should not be used.
- If element/relation is described in meta-model, it does exist and should be used with the same name/meaning and under the same constraints as they are defined in the diagrams.

Specific stereotype `<<SWAElement>>` is introduced. It is used only within meta-model UML diagrams, not in the other parts of UML. It is possible to have more `<<SWAxxx>>` stereotypes for the future.

Here is the part of the meta-model, related to SWA breakdown.



The following points must be put on focus:

- One *System* consists of several *Components*.
- One *Component* consists of several *Blocks*. Some components may have association with other components, representing tree of such. The whole tree can be nested with one *ManagedComponent*. From other side *ManagedComponents* are never nested within each other nor have associations in between.
- One *Block* consists of several *Units*. Blocks are never deployed neither reused separately. They are intended to put particular functionality in one place in order to be maintained easily → factored out, located at particular memory, excluded by conditional compilation etc.

Example: TBD

- *Units* are the smallest elements in SWA. They are subject of unit testing with VectorCAST.
- One *Component* can have *Pattern* associated with it. The following use-cases are possible:
  - MVC + DPConn can be used by SWCs (typically) and CDDs (rarely). Ex: CDD can be a gateway between two CAN networks.
  - MVC can be used when we have tree of *components*, nested within one *ManagedComponent*. Ex. SWC which is CTRL, looking from outside, but inside has a sub-systems which components also are models, views and controllers. Typically segment and raster GUI.

### 6.1.1 BSW – Basic Software

#### 6.1.2 BSWExt – Basic Software Extensions

The need of such extensions is caused by the next factors:

- Autosar requires some parts of BSW to be completed by the end user project.
- Autosar does not provide all needed services, so they needs to be implemented somewhere.

SWA defines BSWExt category this way:

- Every BSW component that needs to be extended will have complementary part called *xxExt*. Ex. EcuM → EcuMExt, Tm → TmExt etc.
- Logically they are located in the layer/package where the original BSW component is.
- Physically, on the local disc/ configuration management it is recommended to be separated in order not to mix our code with the delivered one from Vector/Spansion.
- There are two different kinds of BSW extensions:
  - Implementation of so called "call-out" functions, defined by the Autosar specification. Ex. EcuMExt, BswMExt
  - Implementation of additional services. Ex. Tm → TmExt
    - Exposed to RTE
    - Not-exposed to RTE

#### 6.1.3 LIB - Libraries for shared use

This chapter clarifies for which kind of libraries is this pattern and how to deal with the shared libraries in general.

Recent DI projects need various kinds of libraries. Making one and only set of libraries used by all SW components is utopia:

- Internally developed components might be enforced to use the same libraries, 3<sup>rd</sup> party ones not. If we change them we will lose the warranty and will expose the project to risk of not tested parts of the system.
- Some libraries can be suitable for one part of the system, but not suitable for other one. Typical case is usage of fixed pointer math in automotive part of the cluster, which is fast and precise to satisfy the requirements, which is not the case for the same math functions on the GUI part, where we need single or double floating point precision.
- Some components are coming from platforms and/or 3<sup>rd</sup> parties in form of object code and we cannot link our libraries. We need to provide the required ones. For ex. CRT library.

In the way shared libraries are form of a good reuse, in the same way they can be a shared disaster. Wrong library leads to systematic issues in many places in the system → quality of calculations, performance and memory usage.

The following library categories can be observed in different projects nowadays:

- CRT library functions
  - Frequently used memory and string operations
  - Trivial math operations
- Crypto libs
- Compression/decompression
- Digital filters
- Image processing
- Text processing
- Date-time processing
- Supporting OOP/OOD in C
- C++ templates → standard and home-made
- Helper C++ classes

More will appear for the future, considering the tendency to merge DI and Audio/Infotainment. Open-source usage influences this as well.

The goal of SWA is to ensure the minimum set of frequently used functions for shared use to cover the majority of the use-cases for the automotive part of the clusters and displays. GUI parts usually have their own set of libraries, coming from the supplier. They are optimized and tested for the particular HMI drawing system and needs to be integrated in the product anyway.

The following rules take place:

- ❖ LIB components in the bookshelf have to be stateless → No RAM used.
- ❖ LIB components must not have circular dependencies, i.e. they should look tree-like (not graph-like).
- ❖ Rudimentary functions for memory operations to be taken from compiler provided libraries. The usage to be limited to memcpy (fixed to move as well), memcmp, memset. LIBs must provide alternative of those and a way to choose which ones to be used by the project. Special diagnostic mode to be implemented to be able to monitor calls to these at run-time. This means that they must be called via macros that can be substituted with different functions and/or instrumented for measurement builds. Optimization for speed is preferred than optimization for size, taking into account Spansion CPUs in the scope of this SWA.
- ❖ String operations are not in the scope of LIB.
  - GUI systems have their requirements/implementations for strings:
    - With specific encoding
    - Standard C strings
    - PASCAL strings
    - Different formats ANSI, UTF-8, UTF-16 ... UNICODE.
  - Individual projects might decide to use particular string library. Even several string libraries might be used in different parts of the project.
- Supported function categories:
  - Digital filters → PT1 and Moving Average are most frequently used. Start from them and add more on demand.
  - Measurement unit conversions for distance, weight, volume and temperature.
  - Date-time functions
  - Interpolation/ extrapolation for 8, 16, 32-bit data
  - Linear and Binary Search, table lookup.
  - Generic helper functions/ macros like bit count, digit count, min/max, etc.
  - FIFO, LIFO, DECK → long term (C and C++ versions needed)
  - Float/fix point arithmetic functions - second priority. Classic Visteon had only fixed point support in the past.

The proposal is to have one big library asset only, based on xJCI LIB\_Math, including several improvements and extensions to satisfy SOSA use-cases. Unused functions will be removed by the linker. During the detailed design this one library to be clearly split in functionalities (ex. interpolation, search...) Collect from the projects and add to the library all frequently used filters like windowed filters, ramp ones (look at trip computer, tank modules for ex.).

It is required to document clearly the range of parameters where library functions work. All traps/precautions must be documented.

It is required to test entirely the functionality in the library:

- 100 % coverage is not enough
- Accuracy testing needed → parallel calculation with high precision to be used as etalon.
- Full error handling verification
- Speed tests, where appropriate

When adopting tests from projects, if the tests are not full, the platform should improve them.

#### **How to deal with absence of a library?**

Project can:

- ask platform
  - bookshelf solution might be available already
  - to create it in favor of the project, if not approved as a platform library
- create on its own, but synchronized with platform how to do it
- purchase it

#### **How to deal with duplication of libraries?**

In general this should be avoided, but not always possible. Two main categories:

- Home-made
  - Try to minimize creation of additional libraries
  - They should not overlap the LIB components, delivered by the platform.
  - They must reuse the existing functionality of LIBs and to extend it. If needed upgrade LIBs first, then create new library.
- 3<sup>rd</sup> party delivered
  - Keep the usage of these libraries only within the domain for which they are supplied. Do not use them outside. Potential replacement must impact only that domain.
  - If not optimal, ask the supplier to fix.

#### **6.1.4 SWC - Application Software Components**

According to roles in the system:

- **Controllers** – implement handling of user and system events (e.g. buttons press, voltage drops, diagnostic modes start/stop). Controllers can change states of models, views and other controllers.  
NB: Communication events related to reception and timeouts of signals are not system ones. They are handled by related models, not by controllers.
- **Models** – produce data and related statuses. Takes their inputs from other models or BSW. Produced data is in standardized units (preferable "SI" measurement system). Common approach is to have periodic task to poll their inputs and refresh outputs inside this task.
- **Views** – present data to the driver. Data is provided by the models or by BSW. Views interact with the BSW, related to presentation. Perform unit transformation and rounding. Most common presentation channels are telltales, gauges, sounds, GUI sub-system and illumination.
- **Connectors** – data and event transformation between two data-pools. RTE is considered as a holder of data-pool and some use-cases are:

---

Copies of this document are uncontrolled

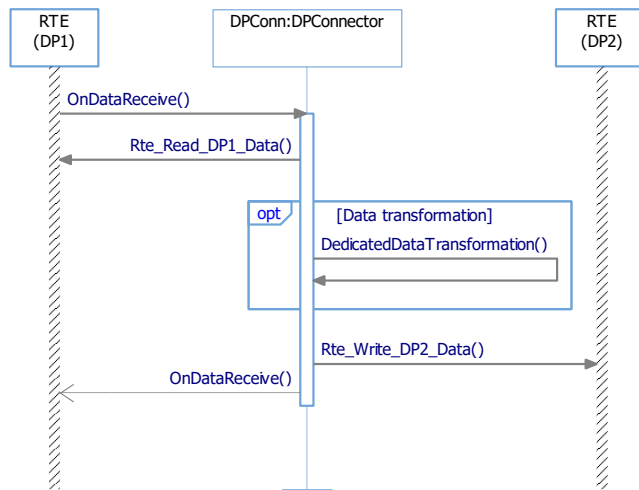
Page 24 of 119



- Between the RTE and GUI specific data-pool (e.g. EB);
- Between the RTE and IPC stack;
- Transmit of RTE data over CAN, LIN, FlexRay with different coding;

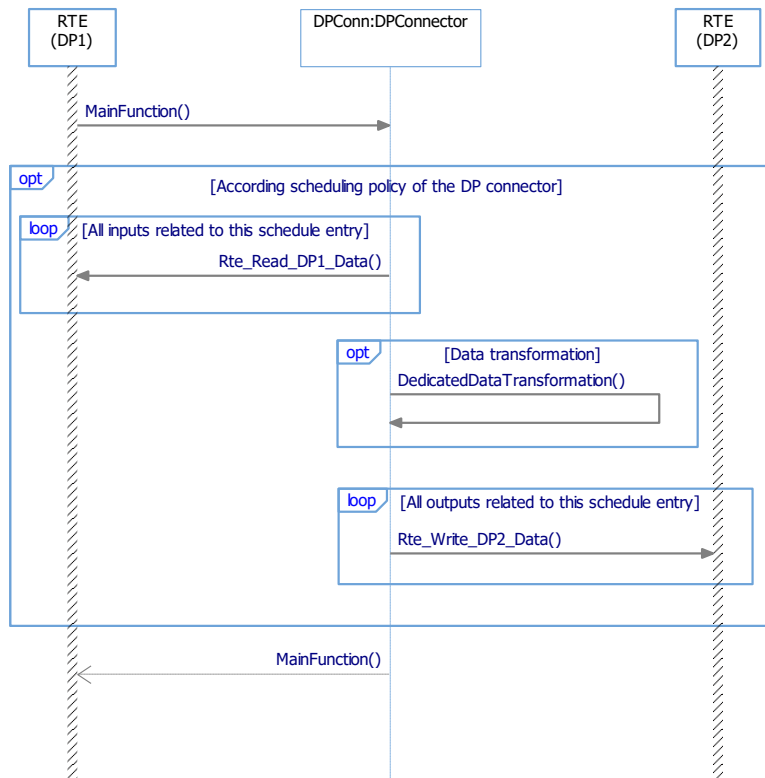
There are few design and implementation approaches recommended for data-pool connectors:

- Handling of Sender/Receiver communication with dedicated runnable:



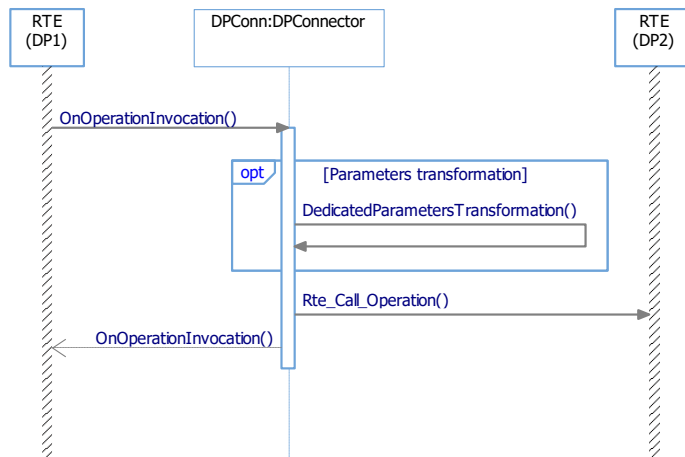
Details	The activation event of the runnable is set to <b>DataReceivedEvent</b> . Runnable implementation Reads/Receives all needed data from the data-pool which trigger the event, performs dedicated data transformation (if needed) and Writes/Sends data to the opposite data-pool.
Use cases	Need of fast data transfer between data-pools. Event based Sender/Receiver.
Pros	Fast reaction on data update
Cons	May increase CPU load in case of task switching and context protection performed by RTE.
Remark	Usage of filter on receiver side may reduce number of activation events. <b>DaVinci</b> tools offer <b>MaskedNewDiffersMaskedOld</b> filter only. Pay attention that the tool insists all receivers of particular data to have one and same filter applied. Do not map the runnable to task (if it is suitable), thus it will be executed in the context of the data publisher and will avoid task switching.

- Handling of Sender/Receiver communication with periodic runnable (**MainFunction**):



Details	The runnable is activated periodically on <b>TimingEvent</b> . Handling of the signals is based on predefined functional groups. Each group has set of triggers (time based and/or status based). When a trigger is fired then all inputs of the group are Read/Receive, needed transformations are performed and related outputs are Write/Send.
Use cases	Need of periodic update of the outputs. Need of additional data consistency, not ensured by their producers. Complex transformation logic (e.g. N inputs produce M outputs). Avoidance of peak CPU load caused by usage of dedicated runnable (first approach).
Pros	Balanced CPU load. Update of outputs without input events.
Cons	Delayed reaction on input changes.
Remark	Timing events may be implements as separate runnables with different periodicity, thus avoiding internal time calculations.

- Handling of synchronous Client/Server communication:



Details	The runnable is activated on <b>OperationInvokedEvent</b> generated by one of related data-pools. Runnable implementation will perform needed parameter(s) transformation and will initiate <b>Call</b> operation to the other data-pool. There is no restriction on how many calls will be placed. One C/S call may produce sequence of related interactions with second data-pool.
Use cases	Need of parameters transformation/reorder between client and server parties. Multiplication of calls. Call extension to sequence of calls.
Pros	Fast transfer of C/S calls to opposite data-pool.
Cons	Not applicable for asynchronous Client/Server when calling component needs <b>AsynchronousServerCallReturnsEvent</b> back.
Remark	Do not map the runnable to task (if it is suitable), thus it will be executed in the context of the caller and will avoid task switching.

- Combination of approaches above will allow transformation of S/R communication to C/S one and vice versa.

According to supplier and internal structure:

- Internal** – components designed and implemented by platform or customer project. They are split into:
  - Native** – stand-alone implementation of a single feature or few connected ones;
  - Composite** – consist of more than one functional block with common behavior or strong dependency between them. Functional blocks follow MVC paradigm, but are not intended to be standalone SWC. Interactions inside such SWC are done via direct function calls.
  - Modeled** components created and generated by modeling software such as *Simulink*, *Matlab* etc. Usually models with complex algorithms.
- Third party** – If the component is created outside the company/project, then the SWC is considered “Third party”. May be provided as source files and/or generation tool or object code. Usually models.

#### 6.1.4.1 MVC

This chapter explains the fundament for reuse, maintenance and system assembly of the application layer SWCs. As a basis MVC paradigm is used (see the presentation first). For simplicity of the explanation we will use the next notation  $\{M|V|C|DP\}[index]$ , where *index* is the number of the instance of the respective model, view, controller or datapool.

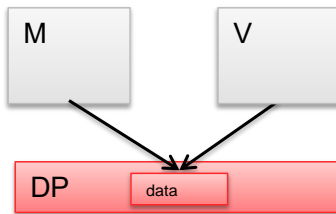
##### 6.1.4.1.1 Connections between SWCs

Every SWC is either a Model or a View or a Controller. Exception is only so-called DP connector component. For this reason we will speak mostly about connections between M, V and C components.

In classical MVC we have direct associations. Example:



In Turing we use Autosar RTE/DP, so direct associations are not allowed. But MVC concept still is kept:



What the important differences are?

Topic	Classic MVC	Turing MVC
Data container	M is data container. It holds and owns the data items that are subject of calculation.	DP is data container. M does not own the data, just calculates it. All memory variables inside M are <i>transitive</i> , i.e. they are used only when M is alive, and has no meaning before and after that.
Life-cycle of the objects	V cannot live, if M is not there or is not operational. Imagine that the association is a pointer to the other object. If the M object is not behind the pointer, V object cannot call it.	V can operate when M is not even assembled in the system. V depends (direct association) on DP, but not on M! DP has default values after POR that are used by V. M can come into the game later on at run-time.
Abstraction of the data producer	None. V object is connected to a particular instance of M. If abstraction is needed, then M must be a <i>remote proxy</i> which will	V uses DP data items. Various producers can be configured to produce the same data. V is completely unaware of that.

	<i>multiplex</i> several other models to provide the same data.	
Abstraction of the way how association is done.	None. Hardcoded one and only way.	RTE/DP is regenerated to abstract the <i>same link</i> in different way: <ul style="list-style-type: none"> <li>• Direct call/direct data access on the same core</li> <li>• Cross cores</li> <li>• Cross virtual address spaces on the same core</li> </ul>
Data protection	None. Must be managed manually.	Protections for preemption, buffering strategy etc. is done by reconfiguring and regenerating RTE w/o changing the connection from component's perspective. Component implementation remains the same.

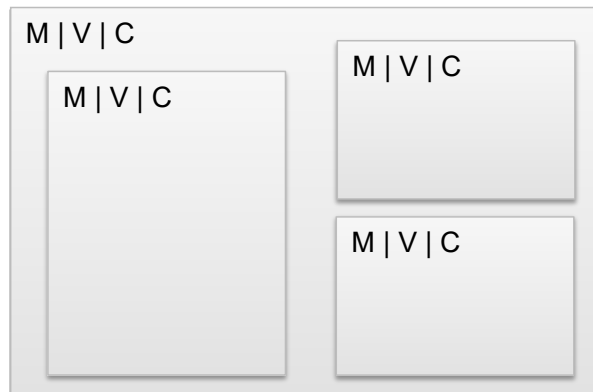
Except data storage aspect, the rest are valid for controllers as well.

Having MVC connections via RTE/DP, opens the next possibilities:

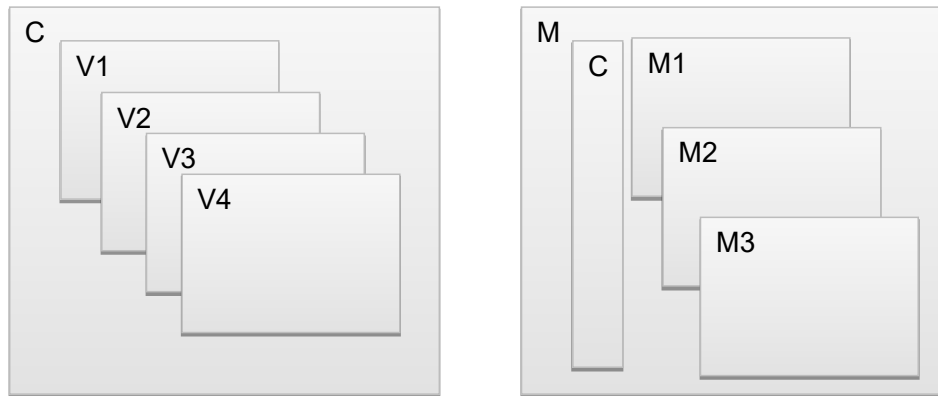
- Design of connections is focused on what the connection should be, not how to do it. The designer thinks what are the needed data items and control commands, exchanged between Ms, Vs and Cs. How they are implemented and reconfigured for single/multicore or how to be used with virtual address spaces, synchronization ... are responsibility of Autosar RTE.
- Individual components can be designed and tested in random order, w/o having available the components on the other side of the connections. This is because RTE instantiates data items and creates stubs for access functions, so there are no pending ends of the connections.

#### 6.1.4.1.2 Hierarchical components

Practically, MVC is used not only at SWC level. Some SWCs are composite. Inside them MVC could be applied, if needed. Theoretically, it looks like this:



Every nested level can be split in MVC components. Examples:



One C and several V are typical for UI systems → e.g. segment LCDs.

Second case could be for example model for speed calculation. Inside you have several models that calculate it via different algorithms → by CAN signal, by Tachograph, by GPS. Also there is a controller, which says under which conditions, which calculation is relevant. Only one active at a time, or all running simultaneously and arbitration is done at the end.

#### 6.1.5 CDD - Complex Device Drivers

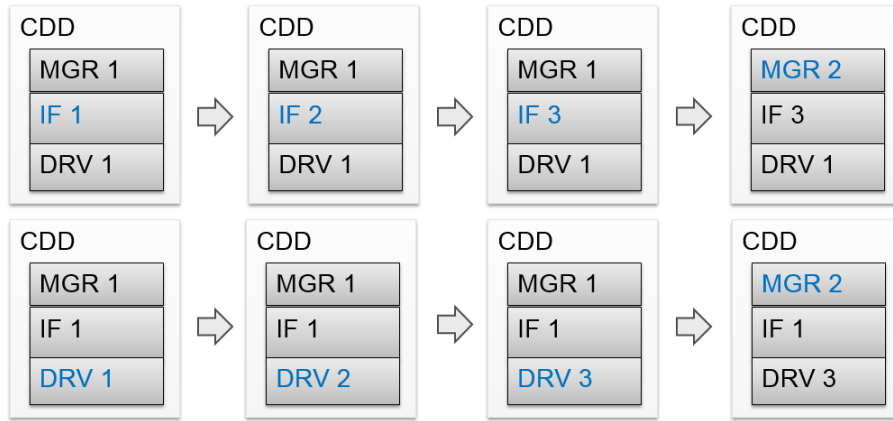
Complex Device Driver pattern is defined by AUTOSAR. Refer to its specification about requirements and constraints.

SWA imposes next additional constraints on provided CDDs and on project specific CDDs.

Most CDDs are split into four parts:

- **CDD** (mandatory) – wrapper of the driver. It maps RTE interfaces to the SRV layer interfaces. This wrapper implements live cycle management of the CDD.
- **MGR** (mandatory) – real implementation of the interface of the CDD. The CDD wrapper uses SRV interfaces to perform requested actions.
- **IF** (mandatory) – HW abstraction part of the CDD. Its functional interfaces are used by SRV part only. Implementation related to CDD live cycle is called by the CDD wrapper. HWAL interacts with related MCALs only. Notifications to the upper SRV part are allowed.
- **DRV** (mandatory) – Microcontroller abstraction of the CDD. Its functional interfaces are used by HWAL part only. Implementation related to CDD live cycle is called by the CDD wrapper. MCAL may interact with MCU periphery and/or other MCALs only. Particular CDD can omit implementation of dedicated MCAL part. In this case its HWAL uses external MCALs. Notifications to the upper HWAL part are allowed.

The same CDD might have variants that are assembled from different parts. Some use-cases:



For communication CDDs, the internal structure is recommended to follow Autosar split in Xcp, Nm, Sm, Tp, If, Drv/Trcv parts.

#### 6.1.6 RES - Resource storage

In this chapter *resource* means piece of information with the following characteristics:

- Kind → Denotes the nature of the resource – ex, sound, text, image, etc.
- Format → The way how the information is physically stored on a media.
- Storage → The media repository for storing the information in a form of database
- Descriptor → Defines the structure of the information for every resource element.
- Resource map → Defines where resource elements are located.
- Interface → Methods to access separate resource elements from the database by using a unique identifier.
- Selector → In case of multiple variants (instances) of resource information into one storage, selector allows to have only one instance active at a time.

The role of the resource storage:

- To provide interface to access separate resource elements, addressed by unique identifiers. Those identifiers are provided by the RES components. The role of this interface is to completely isolate clients from the way information is stored:
  - Location of information must be abstracted (internal flash, external flash, RAM, EEPROM ...)
  - Format of information could be different for different resource elements.
  - Multiple instances (banks) of resources could exist in parallel – run-time selection via interface.
- Resource information should be organized as database by using IDs. This requires internal resource maps to be supported. This ensures that during the life-time of the projects resources can be moved from one media to another media and from one format to another format by exchanging the map and enabling/disabling different converters like de-compressor for ex. The whole logic of the component will remain the same.
- In 90 % of the cases resource database is generated by tools. This remains as a basic principle to ensure fast and safe changes at later project phases.

For every kind of RES components, SW architects decide the particular structure taking into account the next:

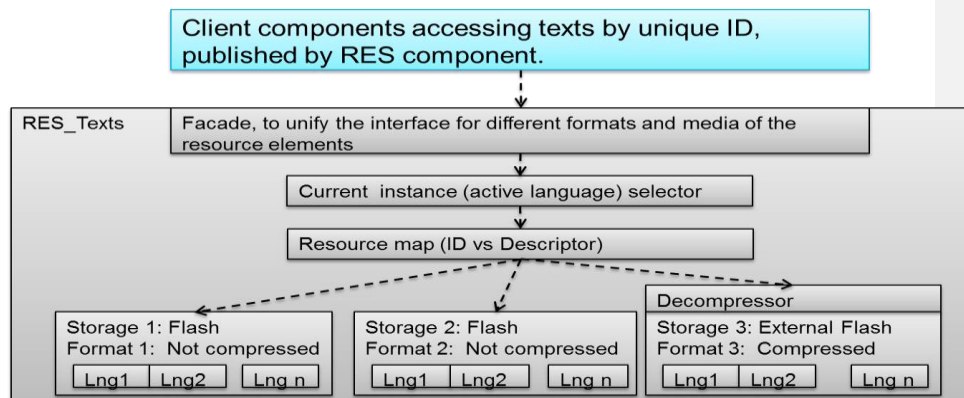
- Storages could be added/removed to the components on project demand.

Copies of this document are uncontrolled

Page 31 of 119

- Resource elements could have various formats, even customer can enforce his own format to be used. This must not change the way how resources are managed/used.
- Conversions like compression/de-compression must be able to be enabled/disabled (or mounted/dismounted).
- Special care to be taken for resource variants:
  - per product lines within the project;
  - per customer brands within products;
- In many cases resource information is:
  - separately flash-able;
  - sometimes it must be re-locatable in memory;

Components of RES type usually serve only one kind of resources. What could be the structure of those? Let take multilingual texts for example:



The benefits (+) and precautions (-) of using RES pattern:

- Numeric IDs
  - + can be accessed from several threads and address spaces, transported over RTE in a safe manner. No issues like with pointers to descriptors.
  - - ID must be kept unique
- Facade:
  - + Increases the reuse
  - - Must be designed to cover all use-cases
- Instance selector:
  - + Flexibility for design and run-time
  - - Protection and default behavior must be selected in case of out-of-bound dynamic selection is tired
- Resource map:
  - + IDs may go to different storages in different product variants
  - - Needs optimization for size and speed for seeking
- Storages:
  - + Number and type can vary depending on the project needs
  - - Consumes physical resources if several storages are used simultaneously - compromise must be made to be used effectively. For example: Overall set of storages in one project is divided into groups per market segments. The storages in one group can go together in the target where they can be selected at run-time. This solves the problem with the place to store them, but requires separate executable to be assembled for each market.



### 6.1.7 RTE - Runtime environment

White: Workproducts  
Green: External supplied tools  
Orange: Home-made tools

- By user request via GUI (java+ SWT) or
  - Via command line in silent mode
- Load **arxml** file, produced by Rhapsody UML model (A2TOM profile). UML model contains all product variants, so during arxml generation particular variant will be selected.
- Source code of TB RTE related files will be renovated according to arxml definition.

During the detailed design of RTE generator it is needed to investigate:

- ARTOP model to internal model, which is serialized to text
- XSLT usage

#### 6.1.7.1 Data items and control commands

RTE is used to manage two major kinds of things:

- *Data items* → Information, subject of elaboration by the SW.
- *Control commands* → Notifications, commands or events, which controls the elaboration.

To be able to distinguish between these two categories, please take this metaphor - 'A factory for producing candies'. What we have here:

- Different materials enter the factory (*various input data*)
- On the production line there are multi-step process to create chocolate, mix with nuts, decorate, etc. So, materials are transformed into different parts for different types of candies. (*input data is transformed into intermediate data items with a dedicated purpose*)
- For every type of candies you have assembly process (*for every type of output data (speedometer, odometer, trip computer) you have a process of elaboration from the intermediate data items*)
- On the production line there are manipulations done over materials: grind, mix, form, bake, .... envelop, pack, send out for sale. (*These are algorithms applied over data like encrypt/decrypt, convert, scale, accumulate, filter etc.*)
- For every manipulation production line executes actions in a sequence (imagine commands for robots): Some thing like Start, Load, Shake 10 min,Unload, Next command etc ... (*In software these are control commands, exchanged between components like Init/DeInit, Reset, Update, Activate/Deactivate etc.*)

Sometimes it is difficult to distinguish between *data items* and *control commands*. Here are some frequently made mistakes:

- RTE supports Client-Server (C/S) and Sender-Receiver (S/R) communication. So, because S/R instantiates variables inside generated RTE, this means that this is *data item*. From other side C/S usually does not instantiate data in RTE, so this represent *control command*. This is wrong, because S/R and C/S are just communication mechanisms. There are other communication mechanisms (in other OS) such like pipes, sockets, mail boxes. This does not change anything in the meaning or usage of *data item* and *control command*. They still exist, but will be managed differently.
- In OEM's specifications, there are wordings, which denote part of the input/output signals of the device as "control signals" for some things. Thus, people start assuming these as *control command*. It is very important to understand that every information that enters the device and goes out of the device is considered as *data items* from SWA perspective. More precisely, the SW can be represented in abstract way as:

[input data] → (calculations) → [intermediate data] → (calculations) → [output data]

Here *control commands* are not shown, because they manage modules that perform calculations.

By *intermediate data* we mean the Data Pool (DP) concept, not all possible memory variables in the SW!

Ex.: Button events are usually denoted by OEMs like display control signals. From SWA perspective this is just incoming data:

- They will be converted by BSW calculations into intermediate queued *data items* into *Data Pool*.
- Some *controllers* (meaning from MVC pattern used; please pay attention to the wording 'controller') will read the data items from the queue and will make a decision which *control commands* to send to which *views*. Please, note that at that moment data item disappears, because it is consumed. This is because of the event nature of the data item. This is not the case with odometer data item in DP, which stays there to be accessed multiple times after its production.

Some hallmarks to distinguish between data items and control commands:

- *Control commands* are pure SWA defined things. They never come with OEM's specs neither with internal specs. They are used to manage the component's behavior.
  - *Controllers* usually emit *control commands*.
  - *Models* might emit *control commands* via *Observer* pattern.
- *Data items* are usually defined by OEMs and our internal requirement specs (ex. manufacturing one). Most of them are associated with resolution and measurement unit or represent enumerator of values.

#### 6.1.7.2 Component interactions

There are 3 important aspects of component interaction that must be clearly understood, before making design of particular SW system.

##### Component to Component

TODO

##### Component to Datapool

TODO

##### Conveyer

TODO

## 6.2 Naming convention

Unique tokens are maintained in SIR form. They are used to derive identifiers from them.

During the SWA workshop 14 to 17-Apr-2015 in Sofia, Bulgaria, the next convention was approved:

### UML: <<SharedPackage>>

*SharedPackageName* → [*Architecture*][*Variant*]*<UniqueToken>*

*Architecture* is the name of the SW architecture. Omitted by default for Turing SWA.

*Variant* is the name of the variant of the shared package.

*UniqueToken* is one token form the database in SIR.

### UML: <<DesignUnit>>

*<SharedPackageName><FunctionalName>*[*Extension*]

*FunctionalName*

Copies of this document are uncontrolled

Page 35 of 119

- For SWC, the functional name will use the postfix “Mdl”, “View” or “Ctrl” for Models, Views and Controllers
- For BSW, the functional name will use the ones defined by Autosar when relevant (Mgr for Manager, If, Drv, Tp...). New ones are created by the architecture team where needed.

**Extension** Not used in Turing SWA.

PsaIpcMgr, Psa3008IpcMgr, StpMgr, BmwStpMgr, PsaStpMgr,  
PsaIpcIf, Psa3008IpcIf, Psa3008IpcIf2, StpIf, StpDrv  
OdoMdl, PsaTtCtrl, TripAfcMdl, TripDteMdl, PtrView, PtrBaseView

For CDDs the next is applied:

[Variant]<UniqueToken>Cdd	is the name in DaVinci model and in SIR
[Variant]<UniqueToken>MgrCdd	is the name of the former SRV layer of CDD
[Variant]<UniqueToken>IfCdd	is the name of the former HWAL layer of CDD
[Variant]<UniqueToken>DrvCdd	is the name of the former MCAL layer of CDD

The functions inside the source files do not have Cdd token, except those generated for RTE interfaces.

KmtCdd	PsaKmtCdd
KmtMgrCdd	PsaKmtMgrCdd
KmtIfCdd	PsaKmtIfCdd
KmtDrvCdd	PsaKmtDrvCdd

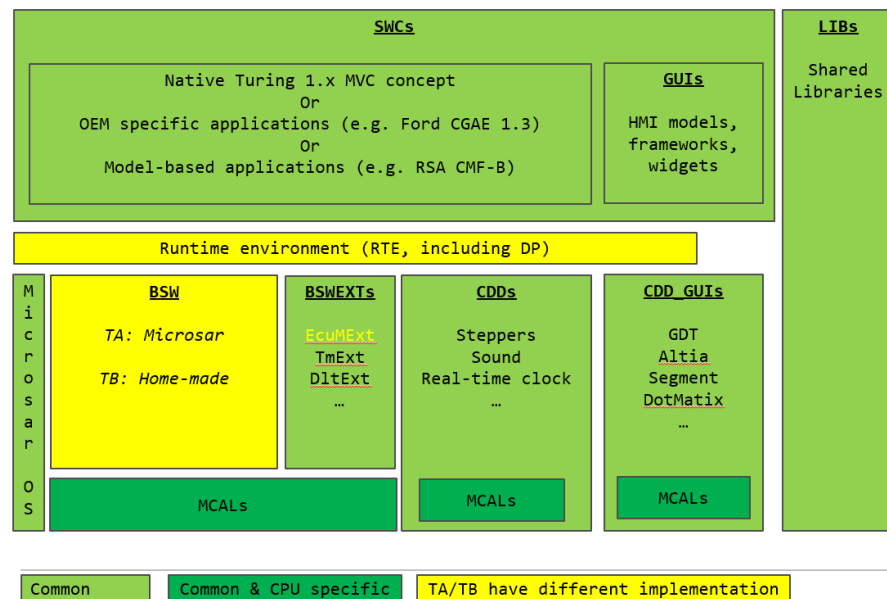
For the communication CDDs:

[Variant]<UniqueToken>Cdd	is the name in DaVinci model and in SIR
[Variant]<UniqueToken>XcpCdd	is the name of the XCP part
[Variant]<UniqueToken>NmCdd	is the name of the Network Management
[Variant]<UniqueToken>TpCdd	is the name of the Transport Protocol
[Variant]<UniqueToken>SmCdd	is the name of the State Manager
[Variant]<UniqueToken>DrvCdd	is the name of the former MCAL layer of CDD
[Variant]<UniqueToken>TrcvCdd	if transceiver is used

IpcCdd  
IpcXcpCdd  
IpcNmCdd  
IpcTpCdd  
IpcSmCdd  
IpcDrvCdd

### 6.3 General diagram

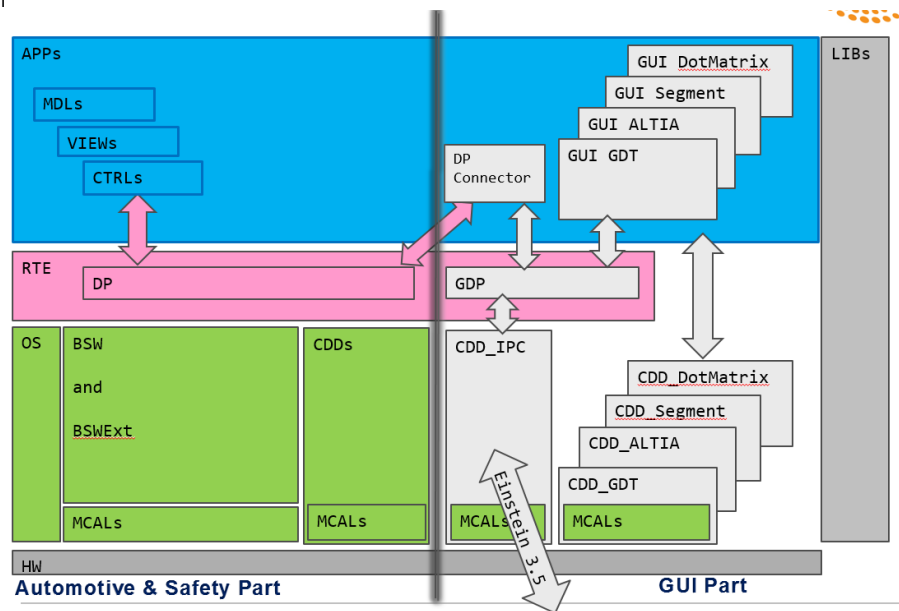
Reuse between TA and TB:



The following types of construction elements exist:

- *Software System* is the whole executable that is going to be created. It is a subject of build process.
- *Component categories* are sets of components with same purpose, located in the same "layer":
  - **BSW** → Autosar and non-Autosar basic SW components
    - **OS** → Operating System MICROSAR OS
    - **TA BSW** → Autosar basic SW components, used in TA
    - **TB BSW** → Non-Autosar basic SW components, used in TB
  - **BSWEXT** → Extensions of BSW components used in TA and in TB
  - **SWC**
    - **MDL** → Models in MVC terms
    - **VIEW** → Views in MVC terms
    - **CTRL** → Controllers in MVC terms
    - **DPConn** → Data pool connector for GUI
  - **RTE** → Runtime environment
    - **VFB** → Virtual function bus. Controlling signals.
    - **DP** → Main data storage for shared use
    - **GDP** → GUI specific datapool (usually generated)
  - **CDD**
    - **MGR** → Service
    - **If** → HW abstraction
    - **DRV** → MCU abstraction
  - **RES** → Specific pattern to handle resources(text, sound, graphics)
  - **LIB** → Libraries for shared reuse

The important point is that LIB components do not depend on the rest of the system. They are treated as living in a parallel layer, which can be called by any component from the main layer.



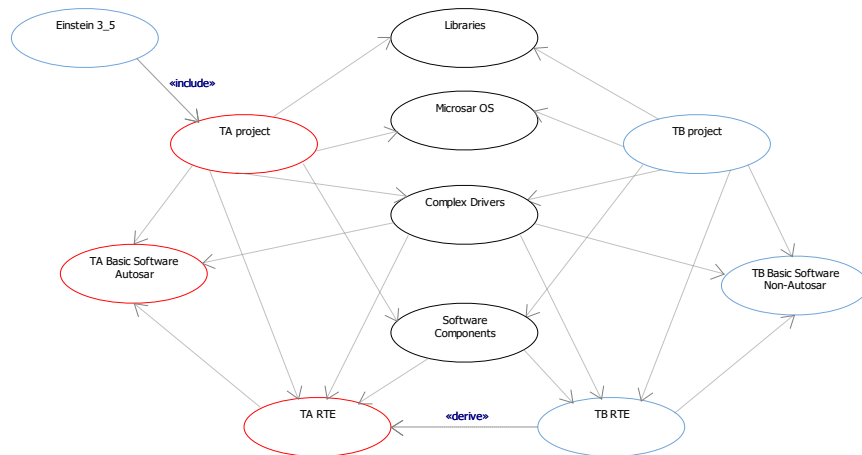
This diagram depicts several important elements at once. It is intended to provide high level view of components that might be used in system assembly. Not all these components will be put together. Which ones exactly depends on the business case of the particular project that will use the platform.

Please, pay attention to these elements:

- There is *Automotive Part* → it, or part of it can be safety; And *GUI Part* which is non-safety by default (this does not mean that user project cannot make it safety, it is just not in the scope of the platform).
- *MVC + Datapool concept (DP)* is used to ensure usage of several GUI systems with the same automotive part.
  - GUI part might be remote via IPC:
    - It is possible to use IPC to another core on the same CPU.
    - It is possible to use IPC to another CPU, where GUI system is located.
    - It is possible to use IPC to another CPU, where GUI system is located and at the same time to have another GUI system in the primary CPU/Core.
  - Data pool (DP) might be:
    - Single, used by all MVCs and GUIs → This is suitable when project has full control of the design of the system, including HMI parts, and wants to minimize the footprint and to increase the performance.
    - Double, with connector in between → this is needed especially when:
      - OEMs provides the specification of so-called GUI DP, which must be updated regularly. The DP part should remain in order not to change MVC components that are producing it.
      - Primary system located on the first CPU must be done once and over the time, different product variants have different GUI systems on secondary GUI CPU. Here, the specification covers the entire feature scope and GUI systems implement parts of it.

The goal of the SWA is to keep as much as possible options for system assembly in TA and TB variants. Where there are limitations they will be documented. Also the places where it is possible to reuse components in other architectures will be marked.

### 6.3.1 TA and TB use-cases



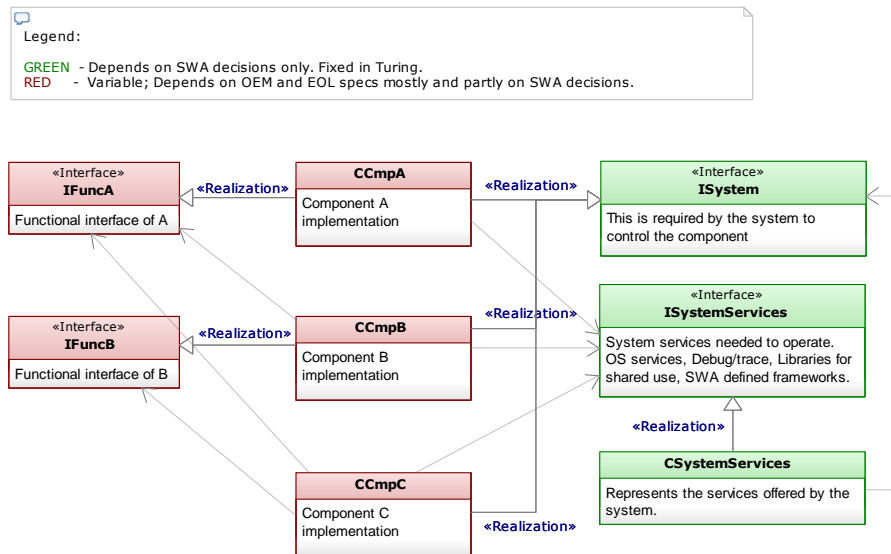
Arrows w/o stereotype mean "usage". Black ovals in the middle are the same for TA and TB. TB RTE is derived from TA RTE (a sub-set of it). Einstein3.5 projects reuse directly one TA project.

## 6.4 System interfaces concept

### 6.4.1 System vs Functional interfaces

This SWA introduces the concept of separation of *System Interfaces* from *Functional interfaces*. These are two different categories of interfaces, which must be understood clearly, because they are basis of reuse and system assembly.

The next diagrams are conceptual and all names inside are for illustration only:



Here we have 3 classes representing 3 components → in red.  
 CCmpA and CCmpB classes realize own *Functional interfaces IFunCA and IFunCB* → in red.  
 The class CCmpC just uses them (depends on them).

All the 3 classes depend on system services (in green) to be able to call them during the calculations.

Also they provide ISystem interface (in green) which is called by the system environment. This is needed to control the behavior of the components from system perspective.

The green parts represent infrastructure, provided by the system to components:

- Decided by SW architects without relation to the project specifications.
- Done once and for all, reused by several projects, following the Turing SWA.
- Reuse is based on the fact that ISystemServices and ISystem are the same:
  - Across product variants within the same project
  - Across several projects

The red parts depends on OEM, EOL, etc. specifications that vary over the development time of the projects:

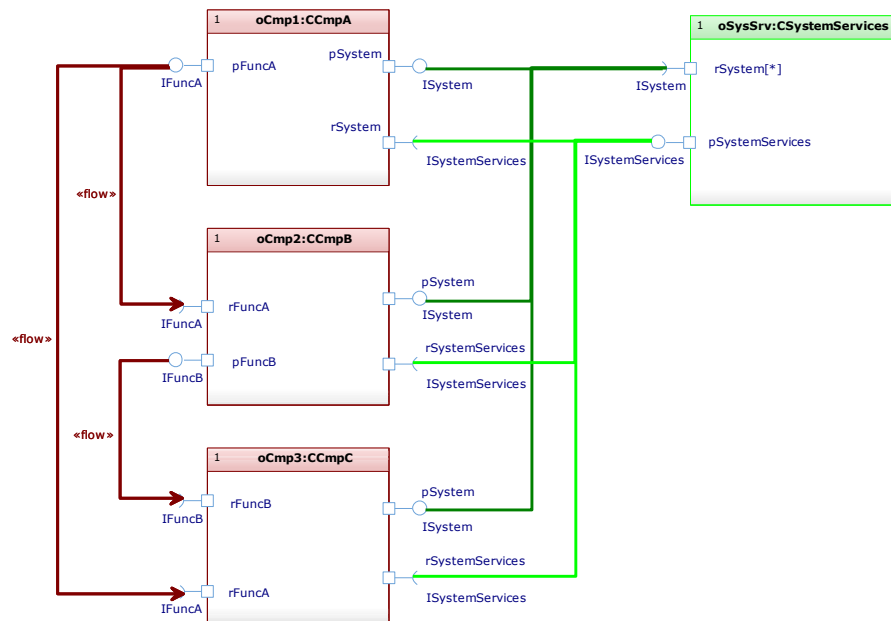
- There is curve of stabilization of those → depends on the stability of the specs.
- They might be able to be standardized within one OEM for one product line. Global unification is either not possible/feasible or not desired by OEMs.

If there is a component, which functional interface becomes specification independent, this means that it can be reused and:

- is candidate to become part of system services
- or at least to be part of a library that is reused across several projects for same OEM.

Here is an object diagram → a particular system out of the class diagram above:





So, every component has two kinds of links:

- Links to the system via *System Interfaces*.
- Links to other components via *Functional interfaces*.

When one needs to plug a component to a particular system, should ask following questions:

- Are all *System Interfaces* used by that component available there? What is missing exactly?
- Are the existing *System Interfaces* compatible, not only from static perspective, but also from dynamic one?
- What are the *Required Functional Interfaces* of that component? How they will be resolved in the new system?
- Who will require the *Provided Functional Interfaces* of that component?

Some typical system interfaces for Turing SWA come from here:

- DltExt → Debug/trace/Assert API
- TmExt → Time services
- MathLib → Conversions, math functions, digital filters
- MemLib → Memory operations, mapped to CRT standard ones

Some typical functional interfaces:

- Com → CAN, LIN, MOST signals
- IoHwAb → Digital and Analog Inputs
- NvM → Parameters for algorithms, settings storages
- Models → Calculated magnitudes like speed, tank content, etc.

Examples:

Copies of this document are uncontrolled

Page 41 of 119

- ❖ The component might use particular standard library (like STL in C++ for example). This is a *System Interface* dependency. When moved to another system, STL might not be there or its version could be incompatible.
- ❖ Model for Trip Computer calculation usually relies on Odometer and Speedometer models. In the object diagram above this is like CCmpC, which needs inputs from CCmpA and CCmpB. Obviously, when moved, its *Functional Dependencies* (dependencies on *Functional Interfaces*) must be resolved in order to work. So, other components have to implement the needed required interfaces.

TODO: Notes from the BSW interface discussion:

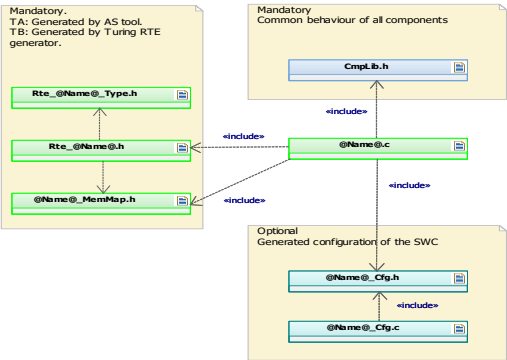
BSW Ex API

- SWCs/CDDs - interfaces to RTE are defined
  - BSW - we develop BSW to fill gaps in AS BSW
  - BSW Ex - extensions to AS BSW components
  - Options
    - a) no standard interface
    - b) different interface for BSW and BSW Ex
    - c) same interfaces for BSW and BSW Ex
  - We chose (c)
    - void Init(void), void DeInit(void), StdReturn Sleep, StdReturn Wakeup, Periodic
    - This matches AUTOSAR so it must be maintained for Turing A and Turing B
  - Naming convention
    - Task != periodic runnable
    - Periodic runnable = MainFunction
    - CmpName\_<MeaningfulName>, CmpName\_MainFunctionRx, CmpName\_MainFunctionTx,
    - <MeaningfulName> - MainFunction for runnable, change only used for multiple periodic
- runnables
- Ex: CmpName\_MainFunction, CmpName\_OdoFunction
- Return type
    - Do we want return type to avoid void?
    - E\_OK, E\_NOT\_OK - This is the same set as AUTOSAR BSW components
    - We will not extend to include E\_NOT\_READY for BSW and BSW Ex
  - Revisit Return Type for Init and DeInit at Turing 1.4 (or Decentralized Services if this is integrated)
    - E\_NOT\_OK
      - Sleep - component cannot sleep, actions can be reset device, ignore, retry, wakeup system, etc
      - Wakeup - component cannot wakeup, actions can be reset device, ignore, retry, maintain sleep, etc

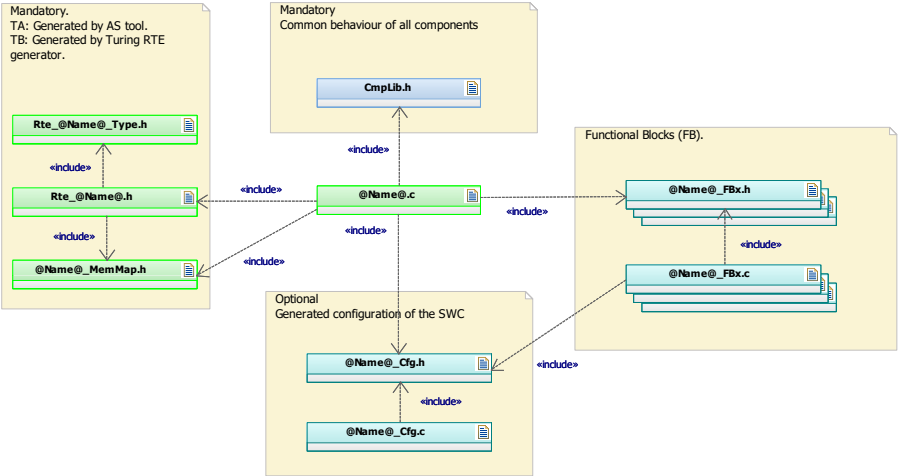
## 6.5 File structure

### 6.5.1 SWC file structure

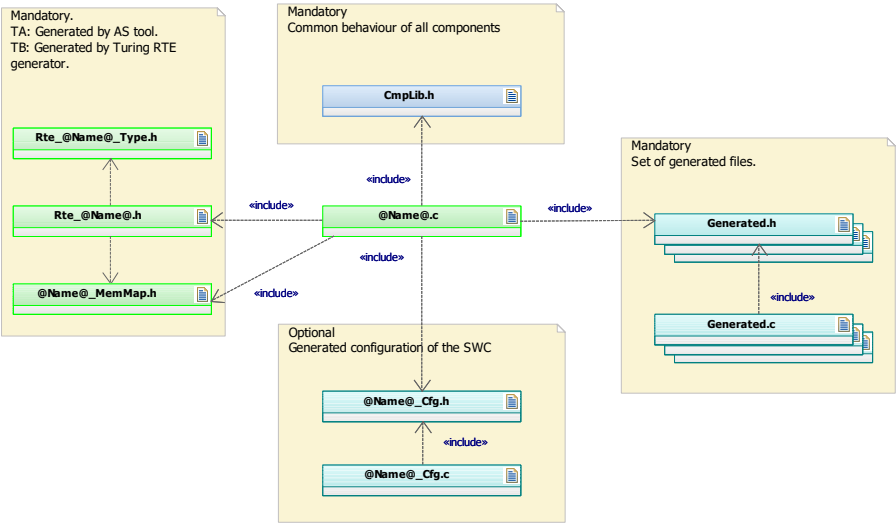
Native:



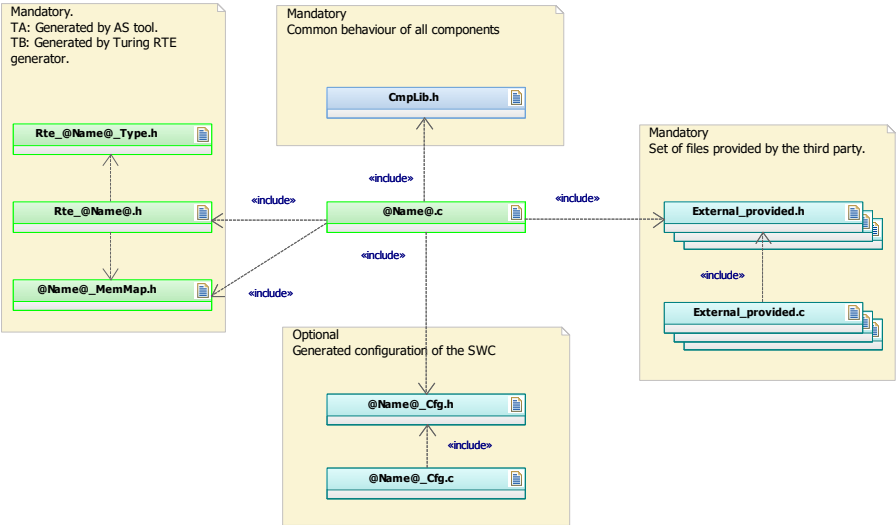
Composite:



Modeled:



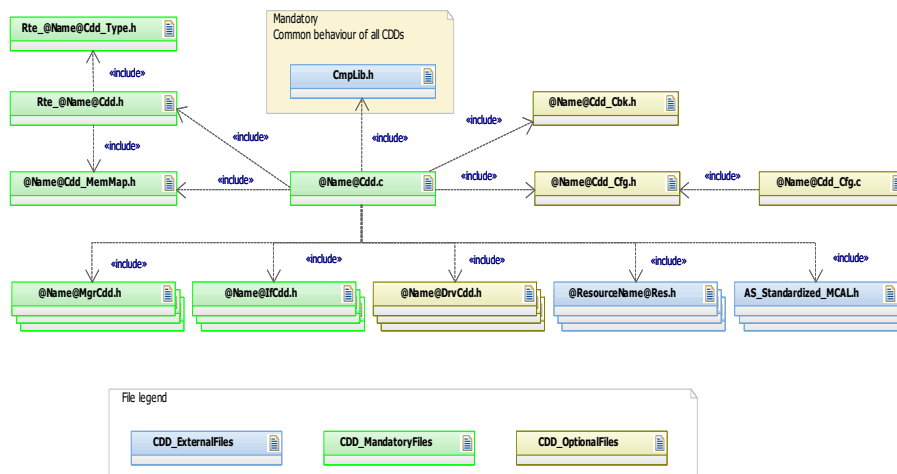
Third party:

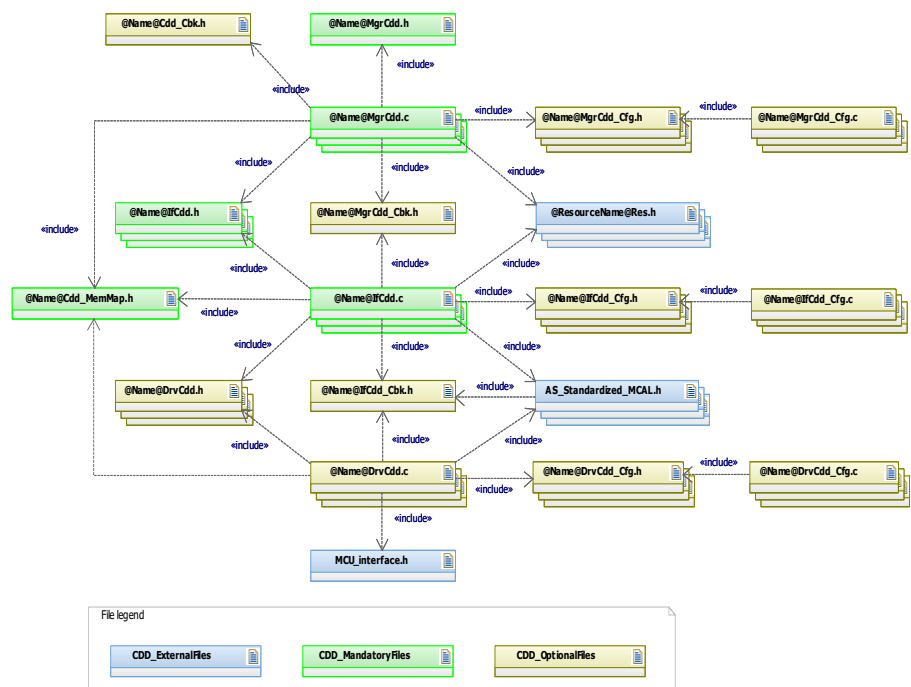


Those are files constructing single SWC and needed to integrate it. Their definition do not impose any constraint to the configuration management.

- Mandatory files:
  - **@Name@.c** is applicable to all kinds SWCs:
    - Implementation of component life-cycle → *CCmp* instance.
    - Feature(s) implementation for **Native** SWCs.
    - Wrapper and call-outs implementation for **Third party** SWCs.
    - Functional block's connections for **Composite** SWCs.
    - As long as this file is one and same in different projects and builds, the component is reused (considered one and the same). If there is change in its implementation, then this is component variant or a different component.
  - **@Name@\_MemMap.h** and **Rte\_@Name@.h** are applicable to all kinds of SWCs - component connectors to the system. May differ between projects and builds inside one project. Generated by AUTOSAR system configuration tool or another configurator in case of TB build.
  - **Rte\_@Name@\_Type.h** - types definitions related to the SWC. Applicable to all kinds of SWCs.
  - **<External\_provided>.h** and **<External\_provided>.c** - set of files delivered by the provider of the component. Applicable to **Third party** SWCs.
  - **<Generated>.h** and **<Generated>.c** - set of files generated by modeling tool. Applicable to generated SWCs.
  - **@Name@.arxml** and **@Name@.instancer** - TA description of the SWCs and Instancer file generated from UML. Applicable to all kinds SWCs.
- Optional files:
  - **@Name@\_Cfg.h** and **@Name@\_Cfg.c** - configuration of the component. May differ between projects and builds inside one project. Manually coded or generated by a dedicated tool. Applicable to native SWCs.

### 6.5.2 CDD file structure





CDDs are using common part to access micro registers. Those files should be shared between all lower level components.  
Common file named Bsw\_cfg.h could be used to keep defines used inside Visteon components but configured and generated from the BSW.  
Example:  
Peripheral frequency used inside the StpCdd to generate Pwm is configured inside the standart AS MCAL Mcu.

6.6 Types

6.6.1 Standard types usage

The following decision was taken:

D10	<b>Standard types</b>
Status	Approved
Category	LIB/Types
Description	All developed components will use the TA BSW types → even above the RTE. This includes TB variant as well.
Rationale	Lessons learnt from MQB, Edison etc. This is the same as the decision as MFA2 project.
Comments	Rules how to solve conflicts with external types need to be defined.

Here are details about how to apply it.

R.TYPES.001

The types are supposed to be used as defined in *Platform\_Types.h*

```
typedef unsigned char    boolean;      /* for use with TRUE/FALSE */

typedef signed char      sint8;        /*      -128 .. +127      */
typedef unsigned char    uint8;        /*           0 .. 255      */
typedef signed short     sint16;       /*    -32768 .. +32767    */
typedef unsigned short   uint16;       /*           0 .. 65535    */
typedef signed long      sint32;       /* -2147483648 .. +2147483647 */
typedef unsigned long    uint32;       /*           0 .. 4294967295 */
typedef signed long long sint64;       /* \brief 64-bit unsigned integer */
typedef unsigned long long uint64;

typedef float            float32;
typedef double           float64;
```

Autosar does not allow to modify this file.

The *\_least* variant of the types are not recommended to be used in derived projects.

The rest of the types will be used by SWCs, CDDs, LIBs and all BSWEXTs we will make for the future. In that case MISRA would impose the systematic usage of cast from uint8 to UInt8 with AUTOSAR 3.x. In AUTOSAR 4.x only *uintxx* types will be used, so no risk. Such MISRA warnings should be globally justified and/or suppressed in the code.

This rule means that the RTE types, starting with capital letter (like UInt8, Sint32) and repeating the BSW types will not be used.

#### R.TYPES.002

Because we need 64-bit types that are not defined by Autosar we will add them to *Compiler\_Cfg.h* which is visible everywhere via *Std\_Types.h* (see the inclusion graph below).

```
typedef signed long long  sint64;
typedef unsigned long long uint64;
```

In case of having 32-bits *long long* or not supporting it for the particular compiler there are two options:

- Compiler provides another intrinsic type that is 64-bits. Then we will change the typedefs above to use it.
- Project must remove/not take bookshelf components that use sint64/uint64. Or to convert their code to use 2 x 32-bit element structure and respective library functions for 64-bit arithmetic via structures (this capability exist in some compilers). Worst case such library can be created by the project or reused from other projects.

#### R.TYPES.003

Different xJCI projects are using different *global* predefined set of constants for *Std\_ReturnType*. This makes impossible porting from one project to the other. We are imposing the following rules:

- The meaning of user-defined *Std\_ReturnType* constants for global usage is an enumerator, not a bit mask. Thus, unique values must be selected for them.
- No more redefinition of the existing standard constants like *E\_OK* and *E\_NOT\_OK*. They should be used everywhere as defined in *Std\_Types.h*. This means that *S\_OK* and *E\_FAIL* should be removed from code when ported from Gen2 into bookshelf.

- Frequently used constant `S_WAIT`, denoting that component is not ready with the result will be renamed to `E_NOT_READY == 2` and will remain a global constant, i.e. valid for ALL kind of components. Defined in `Compiler_Cfg.h`
- Frequently used constant `E_INVALIDARG`, denoting that function argument was invalid will be renamed with underscore `E_INVALID_ARG` to avoid duplication with Windows definition visible under PC simulation. It will be a global constant with value of 3, i.e. valid for ALL kind of components. Defined in `Compiler_Cfg.h`

**R.TYPES.004**

SWCs and CDDs still will use the standard RTE defined error codes for `Std_ReturnType` for C/S and S/R. No deviation from Autosar definition for those.

Note: This chapter is only about the usage of global standard types. In general, components like SWCs and CDDs can have complex internal structure (Ex. Segment GUI system consists of several sub-components within one SWC). In such cases there are private interfaces, defining private types, which are not exposed to other SWCs and CDDs. So, these types are not defined in RTE.

**R.TYPES.005**

When linked to RTE, SWCs and CDDs specific types that are visible to the other components should be managed in the standard Autosar way, i.e. described in RTE configuration.

Precaution: Do not use dirty tricks to transport arrays over RTE via pointers to arrays or 32-bit integers. Use the Autosar 4.2 RTE features for array definition.

**R.TYPES.006**

There is a lesson learned on A2TOM about library public types. Libraries/Frameworks might need to define some specific types exposed to users. Assuming that libraries shall not include the RTE these libraries need to define such types within their public headers. The concern is that one library type might be needed for an RTE interface at the same time. In that case the type need to be known as well by the RTE which will generate the same type (creates a type duplication!).

The recommended approach to solve the issue is to protect these public types in the library headers with conditional switch:

```
#ifndef _RTE_TYPE_H
//type declarations are here
#endif
```

The macro `_RTE_TYPE_H` is defined in `Rte_Type.h` header file, which is visible via include `Rte_<component>.h` file. It is always included first, then libraries headers are included. Thus duplication of type definitions is avoided and RTE definitions take place.

*TODO: In some xJCI projects like MQB the types are protected one by one:*

```
#if !defined(RTE_TYPE_eStatusType)
#define RTE_TYPE_eStatusType
typedef uint8 eStatusType;
#endif
```

*For manual working it is not so convenient to do it one by one.*

*It is possible to have common solution for manual and automatic code, by dividing the header files of the libraries in two parts: exposed to RTE and not exposed and to use only one conditional define for this. This way we do not have additional header files and number of conditional defines is minimized,*

Copies of this document are uncontrolled



*which significantly improves the speed of static code analyzers which check all possible combination of conditional defines.*

*During the development we will investigate what is the most appropriate way and, if needed, we will change the official proposal for this issue.*

#### **R.TYPES.007**

Different home-made or 3<sup>rd</sup> party code generators frequently comes with their types. It is their responsibility to be RTE compatible, otherwise architects will make the needed wrappers and casts/conversions to be able to work properly. This could lead to CPU/ ROM loss.

#### **R.TYPES.008**

There is specific use-case → CDDs that are generating types out of external descriptions. Ex. SegCdd generates logical enumerators for segment groups used by GUI segment VIEWS.

The rule is that values for such resource handles are part of particular RTE type and must be presented in RTE description for the particular CDD. I.e. its description exists in arxml file. During the realization phase of platform we need to be pragmatic and we can accept few deviations. For example:

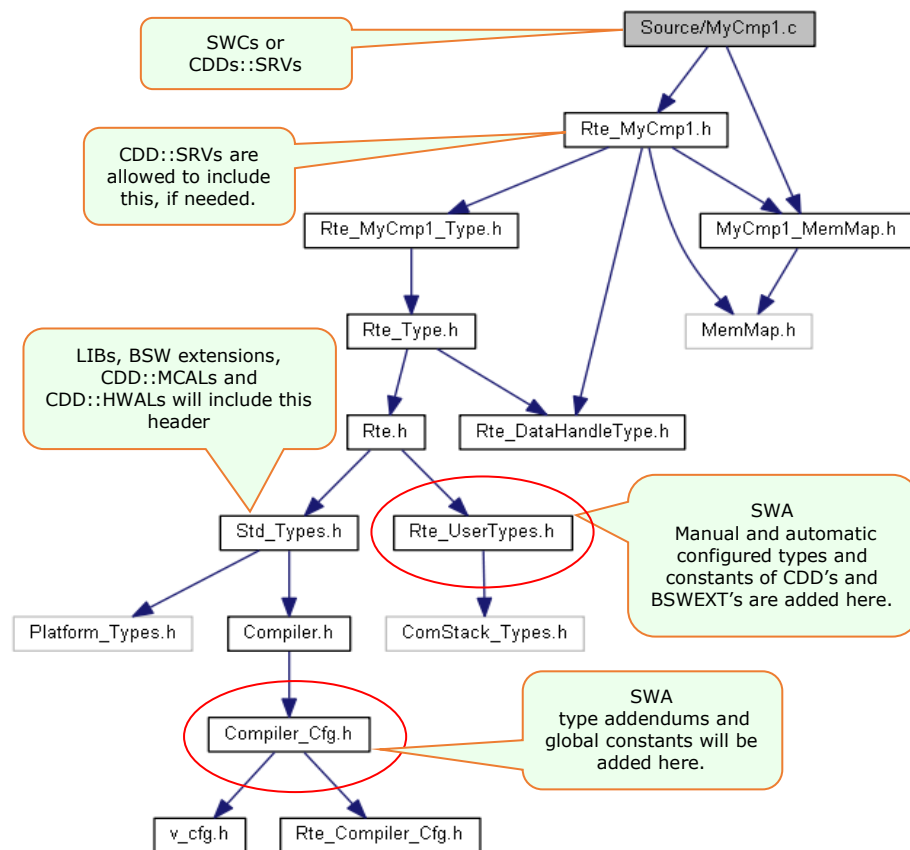
- When reusing an existing solution, we can be open for a deviation.
- We can consider enumerator file (containing typedef and #define) like a library used by SWC and CDD.

#### **R.TYPES.009**

For the basic types, including the additional ones defined by SWA it is necessary to have size checking at compile time. Error must be produced if the type size is not as expected by the platform.

**R.TYPES.010**

When an empty SWC is created with DaVinci, the next include tree is in place:

**6.6.2 User-defined types**

There is a lot of generated and manually configured types and constants that are subject of frequent changes. For example, TmExt timer's handles, Chime's Ids, Segment Group's Ids, Stepper motor's Ids, etc. They have to be made visible to SWC's.

Defining these in arxml to go to DaVinci and to be able to be generated in RTE is significant effort without added value. To avoid this the applied policy is:

- `Rte_UserTypes.h` is the header to be updated to provide those. See the diagram above.
- Include configuration headers of CDDs and BSWEXTs that provide such user-defined types.
- In the future, this file will be generated by the Turing BSW configurator tool.

## 7 Dynamic View

### 7.1 System life-cycle

#### 7.1.1 TA

##### 7.1.1.1 Research EcuM fixed versus flexible

Inside [R5] on "Figure 4.2: Mapping: Phases of fixed EcuM to flexible EcuM" could be seen all the states and transitions defined inside the AS standard and the differences between fixed and flexible EcuM.

The description about the implementation of Vector Informatik could be found inside [R3]. On "Figure 3-2 State Diagram of the EcuM with fixed state machine" the states and transitions could be seen.

###### 7.1.1.1.1 Design alternatives

ALT1: EcuM fixed to be used  
ALT2: EcuM flexible to be used.

###### 7.1.1.1.2 Design choices

In order to have the same implementation for TA and TB, a decision is taken → ALT1. EcuM fixed to be used for TA.

##### 7.1.1.2 Research BswM to be used with EcuM fixed

There is a possibility to use the BswM even when the EcuM fixed is used.

###### 7.1.1.2.1 Design alternatives

ALT1: There are additional states inside ECUM\_STATE\_RUN (see ESM\_Fig.3 and ESM\_Tbl.2). Those states could be implemented with some user logic inside the BswM.

ALT2: Do the state handling manually inside the AppCtrl with the help of CmpLib.

###### 7.1.1.2.2 Design choices

In order to have the same implementation for TA and TB, a decision is taken → ALT2. Do the state handling manually inside the AppCtrl with the help of CmpLib.

##### 7.1.1.3 STARTUP – Initialization

###### PreOSInit

As could be seen at ESM\_Fig.1 the BSW has 2 initialization phases ECUM\_DRIVERINITLIST\_ONE and ECUM\_DRIVERINITLIST\_TWO (*PreOsInit* and *PostOsInit*, before and after the OS start). All MCALs, which do not require OS during their initialization, should be added manually inside this list. The adding is done with the help of the DaVinciCFG tool provided by Vector. The functions EcuM\_AL\_DriverInitZero() or EcuM\_AL\_DriverInitOne(), invoked in this state by EcuM\_Init(). The fictive function BSW\_Mcals\_XXX\_Init() is actually representing the list and the call order, shown in table ESM\_Tbl.1.

The order of initialization is up to the user, but certain rules should be taken into account:

- R1 Ports should be init before the Adc and Pwm;
- R2 Clocks and power domains should be init first;
- R3 IoHwAb should be init last;

Inside EcuM\_AL\_DriverInitZero() it is recommended to do Dem\_PreInit() and Det\_Init(), if respective components are going to be used.

**PostOSInit**

All the BSW components which need OS support during the initialization phase should be initialized inside the ECUM\_DRIVERINITLIST\_TWO.

The fictive function BSW\_HWALS\_XXX\_Init () is actually representing the list and the call order, shown in table ESM\_Tbl.1.

**BswExt\_Wakeup**

All the BswExts wakeup functions to be called in the order shown in ESM\_Tbl.1. The fictive function IBswSystem\_Wakeup() is actually representing the list and the call order.

BSW_Mcals_XXX_Init	BSW_HWALS_XXX_Init	IBswSystem_Wakeup
Mcu_Init(CfgPtr)	DltExt_Init() <sup>(4)</sup>	DltExt_Wakeup()
McuExt_Init()	EcuMExt_EvalWakeupSrc()	TmExt_Wakeup()
Port_Init(CfgPtr_RUN)	FeeExt_Init()	ComExt_Wakeup()
Gpt_Init(CfgPtr)	NvMExt_Init()	NvMExt_Wakeup()
Wdg_Init()	NvM_Init()	PowerExt_Wakeup() //tbd
WdgM_Init()	NvM_ReadAll()	MemTstExt_Wakeup() //tbd
Adc_Init(CfgPtr)	Can_Init(CfgPtr)	WdgMExt_Wakeup()
Icu_Init(CfgPtr)	Can_InitController(CfgPtr)	
Pwm_Init(CfgPtr)	CanTrcv_Hw_Init()	
TmExt_Init()	CanIf_Init(CfgPtr)	
IoHwAb_PowerEnable() <sup>(1)</sup>	CanSM_Init()	
IoHwAb_Init()	CanTp_Init()	
HMemDrv_Init() <sup>(2)</sup>	PduR_Init(CfgPtr)	
	CanNm_Init(CfgPtr)	
	Nm_Init(CfgPtr)	
	Ipdm_Init(CfgPtr)	
	Com_Init(CfgPtr)	
	//tbd Lin_Init() <sup>(3)</sup>	
	Dcm_Init()	
	ComExt_Init()	
	PowerExt_Init() //tbd	
	MemTstExt_Init()// tbd	
	ComM_Init()	

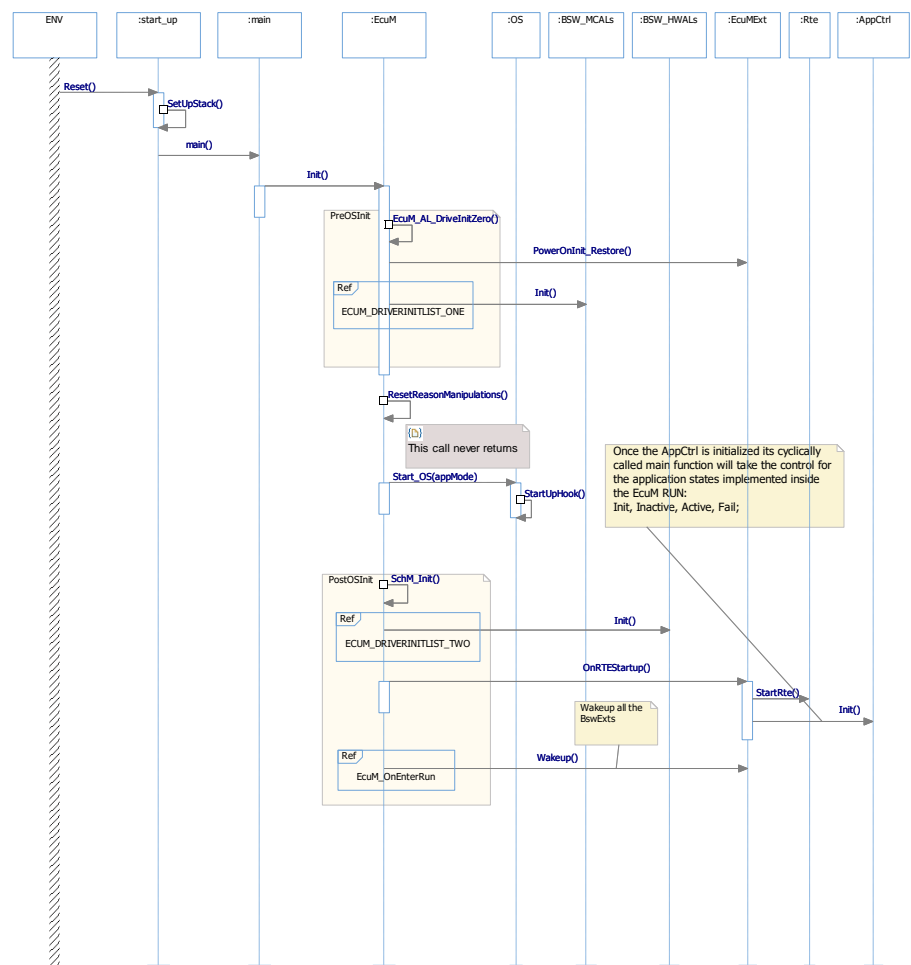
<sup>(1)</sup> Could be also inside the IoHwAb\_Init()

<sup>(2)</sup> If no power pins related to the hyper memories HMemDrv\_Init() could be before the IoHwAb\_Init()

<sup>(3)</sup> LIN init functions could be placed after CAN at that place.

<sup>(4)</sup> DLT will not be used during the MCALS initialization

ESM\_Tbl.1



ESM\_Fig.1: EcuM initialization sequence

7.1.1.4 RUN state

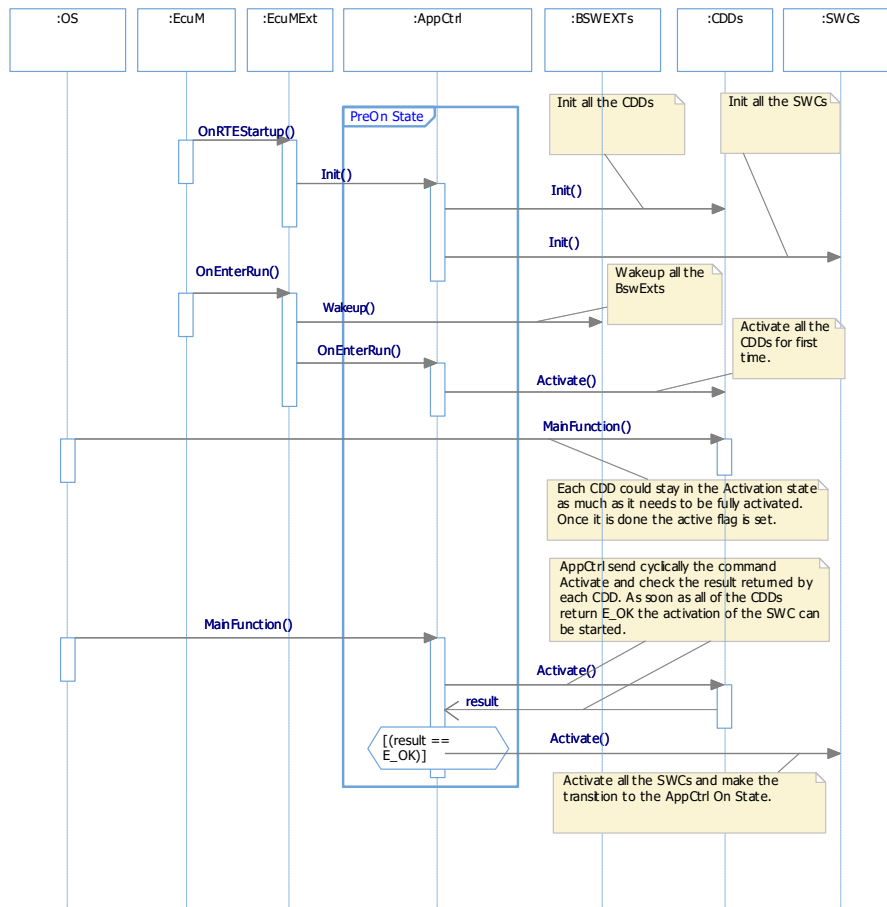
The ECUM\_STATE\_APP\_RUN is extended by AppCtrl with the help of CmpLib to the following states – see the table ESM\_Tbl.2:

AppCtrl(SWC;CDD) states	EcuM States (BSW) - defined by Vector inside [R3]
off	ECUM_STATE_STARTUP ECUM_STATE_APP_POST_RUN ECUM_STATE_PREP_SHUTDOWN ECUM_STATE_GO_SLEEP ECUM_STATE_SLEEP ECUM_STATE_GO_OFF_ONE ECUM_STATE_WAKEUP_VALIDATION

	ECUM_STATE_WAKEUP_REACTION
	ECUM_STATE_WAKEUP_WAKESLEEP
On, Fail, PreOn, PostOn	ECUM_STATE_APP_RUN

ESM\_Tbl.2

Each SWC and CDD has its own instance of CCmp. Detailed initialization sequence is shown at ESM\_Fig.2. There are notifications sent by the EcuM on OnRTStartup() and OnEnterRun() where AppCtrl functions could be attached (see ESM\_Fig.2).



ESM\_Fig.2: Application and CDD initialization sequence

AppCtrl will be the only user of the EcuM and should request/release RUN state. This way it will keep the RUN state as long as needed. It is not needed to request/release POST\_RUN in CDDs neither in AppCtrl, because the job will be done in *PostOn* state.

CDD Init	SWC Init
IicCdd_Init()	SpdMdl_Init()
SndCdd_Init()	EngMdl_Init()
SpCdd_Init()	TripMdl_Init()

GdtCdd_Init()	GdtCtrl_Init()
	WmCtrl_Init()
	SndView_Init()
	TtView_Init()
	PtrView_Init()
	DimView_Init()

ESM\_Tbl.2

The example on ESM\_Fig.2 shows that the SWC activation is waiting all the CDDs to finish with their activation. This could be organized differently depending on the application requirements and implemented inside AppCtrl.

Groups of CDDs and SWCs could be organized in such a manner that their activation is done synchronously. Example:

Group1 (Cdd1, Cdd2, Swc1, Swc2) activated first;

Group2 (Cdd3, Cdd4, Swc3, Swc4) activated once all the members of Group1 are active.

The application initialization is user specific, so the following overall SWC initialization recommendations take place:

- The initialization order should be MDLs than CTRLs than VIEWS.
- MDLs providing data to other MDLs or CTRLs should be initialized first. Example:  
The MDL which cares about the data, shown on the display, should be initialized almost at the end of all MDLs, because it use data from all others.
- MDL managing modes (like car modes) should be initialized before the users of the mode.
- GdtCtrl could be initialized last.

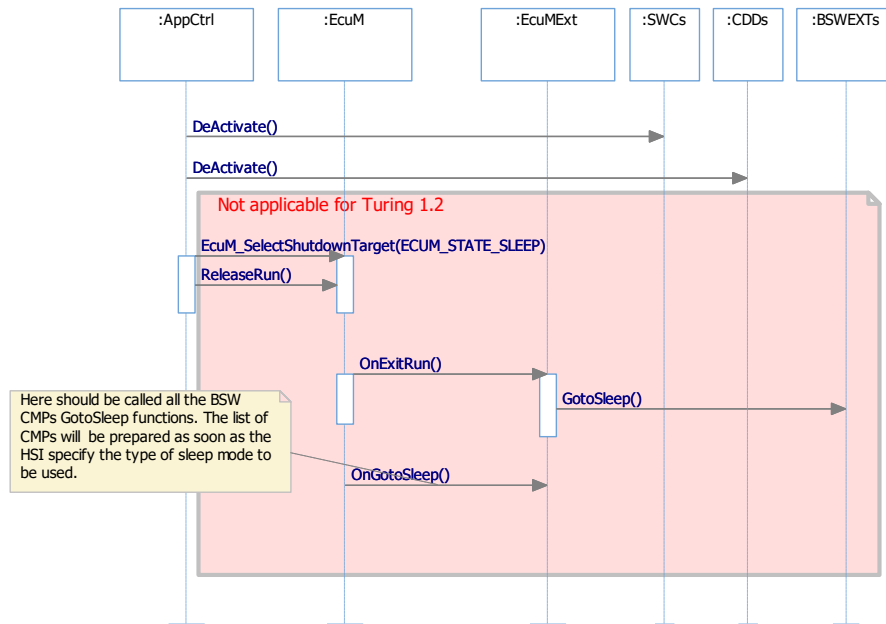
#### 7.1.1.5 POST\_RUN

Not used in TA

#### 7.1.1.6 SLEEP

This is low power consumption mode which does not require reset on exit.

Sequence diagram "seq03\_SM\_Sleep", inside the rhapsody model, describe functions call order and CMP interactions during the going to sleep procedure. There should be a list of BSW CMPs, whose xxx\_GotoSleep() functions should be called. This list should be synchronized to the list called inside the Wakeup state. The EcuM call of EcuM\_OnGoSleep() is used to do the call. The content of this list depends on the type of the low power mode and should be specified inside the HSI. Once it is specified the list creation could be started.



#### 7.1.1.7 SHUTDOWN

ESM\_Fig.3 shows the transition to SHUTDOWN state.

For the initial version of the platform the exit from the low power mode (Shutdown state) will be with a reset. A special care should be taken in the following aspects:

- Data to be stored in the Backup RAM (if any). Should be decided which data to be stored there.
- NVM write all to be finished

If during the going to sleep a condition for wakeup arrives, special care should be taken with the graphical CDD. The known problems are:

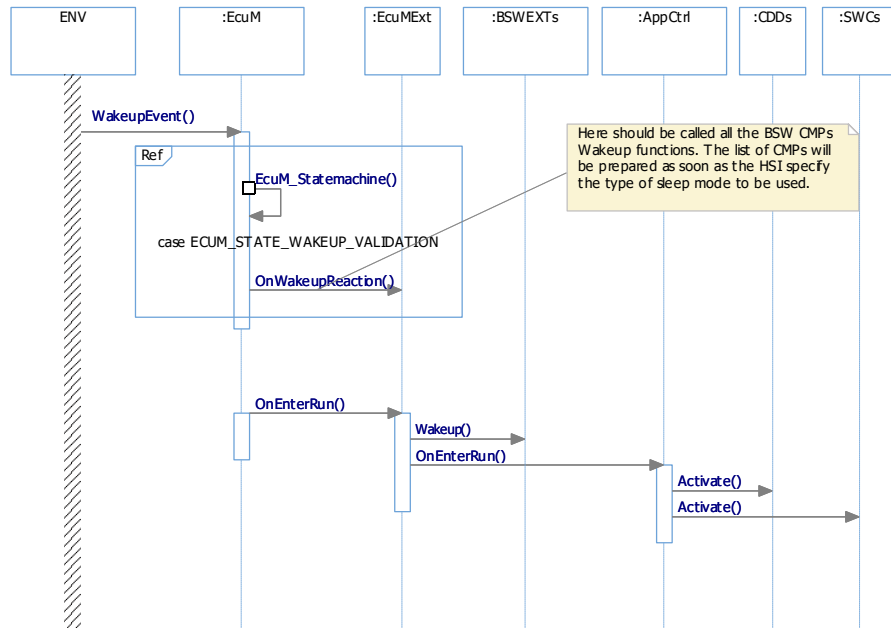
- If there are C++ code constructors usually they are initialized only once after Init and their re-initialization is something complicated, taking a lot of time.
- GUI model could handle a transition from going to Sleep View to a Startup View and event could be generated activating such a transition.

Such a transition is *PostOn* → *PreOn*.

#### 7.1.1.8 WAKEUP

This state is entered on SLEEP exit. Sequence diagram "seq04\_SM\_Wakeup", inside the rhapsody model, describe functions call order and CMP interactions during the wakeup procedure. There should be a list of xxx\_Wakeup() functions which should be called and this list should be synchronized to list called inside the Sleep state.





### 7.1.2 TB

#### Description of the states handled by TB\_EcuM

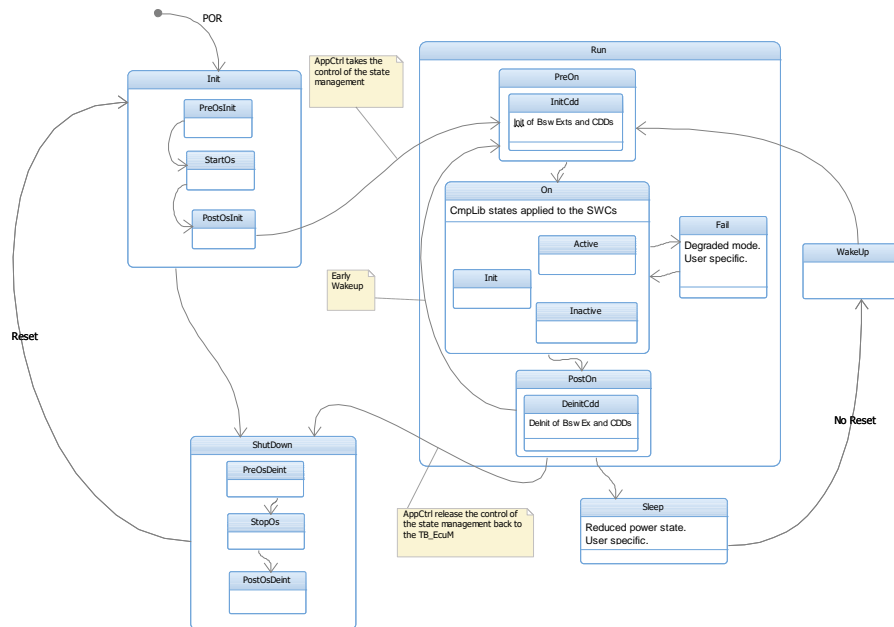
The TB\_EcuM CMP should take care for the following states, functionalities and transitions described below and also shown on the activity diagram ESM\_Fig.3:

- Pre/Post OS BSW CMPs Init;
- Start the OS;
- Pre/Post OS BSW CMPs DeInit;
- Wakeup validation
- Wakeup/Sleep
- Handle the states transitions: Init, WakeUp, Run, Sleep, Shutdown

The purpose of the TB\_EcuM is to implement the functionality of the EcuM fixed described inside the standard.

Requirements and detail info on how it should be implemented could be found inside [R4]. The sequence diagram at ESM\_Fig.1 shows the initialization which is also applicable for TB.

The states handled inside the EcuM are taken from the AS specification and are already described in the section "State management TA". The only difference is that for TB there will be no configuration tool and the BSW CMPs will be home made.



ESM\_Fig.3: Ecu states for TA and TB

#### 7.1.2.1 Description of the states handled by AppCtrl

The AppCtrl purpose is to gather the information from the rest of the system and to release the EcuM RUN state if it is not any more required by the Ecu.

Inside it should handle the states *On*, *Fail*, *PreOn*, *PostOn* see ESM\_Fig.3. It should use CCmp methods of the CMPs to do this.

During the initialization it should ensure that BSWEXTs are initialized then the CDDs and then SWCs (see ESM\_Fig.2). Afterwards BSWEXTs are woken up and CDDs are activated. Once the activation of all CDDs is finished, the activation of the SWCs should be started. Please, note that CDDs need reading of NVM to be finished before leaving the Activation state! The de-activation should be in the reverse order.

**CMP\_STATE\_INIT, CMP\_STATE\_INACTIVE, CMP\_STATE\_ACTIVE, CMP\_STATE\_ACTIVATION, CMP\_STATE\_DEACTIVATION**

The states are described in details inside CmpLib. In the diagrams they are used to display the SWC states handled by AppCtrl.

##### PostOn

The purpose of this state is to be used to deactivate the CDDs after the SWCs. Their deactivation could take longer time. The last thing to do before leaving this state is to call the GotoSleep functions of the BSWEXTs.

##### PreOn

The purpose of this state is to be used to initialize the BSWEXTs and to activate the CDDs before the SWCs. Their activation could take longer time.

##### Fail

TBD

*Inside this state activities should be done to handle over/under voltage or any other special customer requirement.*

#### 7.1.2.2 Description of the IoHwAb initialization

As could be seen on ESM\_Tbl.1 the IoHwAb is initialized before the OS. The reason for this is to have available Vbat, Tpcb or any other critical value available as soon as possible. To make this possible the following inputs are necessary:

- ADC value
- Configuration for filters and converter who will convert the ADC value to physical value.

There are 2 possible scenarios for handling of device variant input which will determine the configuration of the CMPs:

A. Input is coming immediate (ex. from digital input)

**B. Input is coming delayed (ex. from eeprom). This is the most frequent situation.**

In variant A inside the IoHwAb\_Init() configuration should be loaded and all the measurement process could be started on the first call of the task function.

In variant B the information for the variant configuration will be delayed and inside the IoHwAb\_Init() a default configuration for the critical analog inputs (ex. Vbat, Tpcb, ...) should be loaded. In this case on the later cyclic call of the IoHwAb task those values will be treated with the default configuration and once the variant info is available a soft switch to the proper configuration should be ensured.

Special handling of the variant should be done if it is coming from the NVM. Before the Nvm\_ReadAll() call only the min info containing the variant could be read out of the eeprom. If meanwhile a CMP request data from a not initialized IoHwAb, the answer it receives should be *data-not-available*.

#### 7.2 Component life-cycle

In general there are two main approaches to deal with components within SW architecture:

- Components do not have predefined life-cycle nor internal state, representing it.
  - They are treated just a set of public functions and users of the components are calling these functions as they want.
  - Component is supposed to operate properly<sup>©</sup>, in any combination/sequence of calls to its public provided interface. OR
  - The external system that uses the components is supposed always to call the right function in the right time and in the right order.
- Components do have predefined life-cycle and internal state, representing it:
  - They are responsible to protect their life-cycle from abuse, no matter what external system is trying to do via public provided interface. If external system requires proper action regarding the current state of the component then it follows the request, else it rejects.
  - Components still are slaves to the external system → they do not change their life-cycle state by themselves.

As per the global decision above SWCs and CDDs will have predefined life-cycle.

The next use-cases have to be respected:

Different sets of components can operate at different time. System decides this. Component is not allowed to switch off by itself. The only possible self-driven transition would be if component detects RTE event or another command via its functional interface to go to Diagnostic mode. In most of the cases this request also will come from outside.

Some components will need to be configured, based on external factors coming asynchronously and taking time to collect the needed data. Special state has to be ensured for such components. This is not valid for all components, so this configuration state should be optional.

Many components needs to receive configuration from outside to be able to determine how to operate. Typical case are Ford projects where Central Car Configuration is distribute via CAN. It

takes several seconds to collect all the data and to distribute the valid settings to the components. Based on these settings component will know what the right operation to perform is. Another case is when components are waiting EEPROM mirror to be loaded to make a decision. Also some components might require to operate after the first reasonable data comes → they need to wait some time as well. So, in all such cases components are running, but they do not provide their functional outputs (which are their main goal to produce).

Some components will need to release resources they allocated for the normal operation. Optional mechanism to do this must be ensured.

After components are configured and ready to run they need to be in suspended state and to wait for an explicit command to start running. This suspended state might be required later on at run-time. This is needed to be able to chain the components during Init/Wakeup/GotoSleep procedures in order to avoid bad side effects.

Components need to have diagnostic state. This does not include the spying diagnostic methods that will be available in normal operation at run-time. This state is mostly about processing of routine control actions, regarding UDS/ KWP2000 or similar diagnostic protocols.

Rationale for having three outgoing transitions from diagnostic state:

- Some diagnostic requests like calibration will modify the internal state, so, it is needed to reinitialize the component after diagnostic is finished;
- In other cases it is enough just to return where we are:
  - In active state
  - In inactive state

Rationale to complete the diagnostic in two ways:

- From within the periodic task → It is needed when *RoutineControl* is implemented as self-terminated, i.e. no *StopRoutine* is expected to be called. Self-test for ex.
- Either this is *StartRoutine/StopRoutine* or *Freeze/ShortTermAdjustment/Release*

Rationale to have separate command for restarting the component, instead of using initialization command to do the same:

- Command for restart must be operational even when component is inactive. Initialization command is considered completed when component is inactive (off → inactive), thus this command will be rejected, if received in inactive state.
- Application controllers will process restart commands at different time/ place than initialization of the components. If the same initialization command is used for both purposes, it will be confusing and difficult to distinguish in code what exactly is intended for every particular case. Also if modification of the restart behavior is required for some components/use-cases, then both initialization and restart would be affected, so system won't behave normally.

Rationale to have inactive state:

- We need the whole group of components to reach inactive state, i.e. to complete their initialization, then to put them in active mode. This avoids bad side effects caused by partially running sub-systems.
- Tracing component behavior at run-time. Some components needs to be forced not to write in RTE/datapool and data to be manipulated via diagnostic. This can be done with PC simulation.
- Some customer diagnostic requests might require the same.
- Manufacturing usually complain that main functionality is messing the process. Here the components are inactive after system start, so only the needed one for manufacturing can be enabled. The rest won't mess.

The *Life-cycle* for SWCs and CDDs must obey the next requirements:

#### **R.CMP.001**

The *Life-cycle* machine will be implemented in one place and reused by all components that need it. This single place is subject of code protections regarding *Safety* concepts.

Practically *Life-cycle* state-machine will be inherited by components, which will override the actions in different states and transitions.

#### R.CMP.002

After power-on-reset the components must be in OFF state, where they are partially initialized, do not operate, and awaiting external command for initialization.

#### R.CMP.003

Optional state INIT for initialization must be ensured. Here the component is awaiting to collect the needed data for initialization. Collection is ensured by a periodic runnable. At the end of the state component is fully functional. If this state is not needed, then initialization function must be called only once and immediate transition to ON state to be done.

#### R.CMP.004

Optional state DEINIT to perform de-initialization must be ensured. The components is awaiting processes to complete or to release the needed resources, if any. This is ensured by a periodic runnable. At the end of the state component is not functional. If this state is not needed, then de-initialization function is called only once and immediate transition to OFF to be done.

#### R.CMP.005

Normal operation state where component must do the expected functionality is defined as ON state. It is divided in 3 sub-states:

- ACTIVE: Component is running. All degraded modes are operating here as nested states.
- INACTIVE: Doing nothing but the internal component memory context is preserved.
- DIAG: Diagnostic mode. Performs routine control actions and similar. Going out of diagnostic mode should cover the use-cases, listed above.

#### Deployment strategy of this concept:

User projects not always are following the offered architectural concepts from platforms. For example when OEM provides the architecture of the application layer. So, it is not possible to make mandatory the described component life-cycle model.

From other side, the platform will provide all the BSW + CDDs in case of TA and TB. These parts are really interested of having this concept in place. So, platform can implement and test the concept, then to deliver it with instructions how to be used at user project side. Here are the rules:

- SWCs and CDDs made by the platform are using this life-cycle concept/management.
- User projects must follow it, if they create their own CDDs.
- User projects might not use it at SWC level, if they do not want it.
- User projects must apply this concept fully (not only to some SWCs) if they decide to use it. This is related also to usage of application controllers. Check the respective chapter.

### 7.3 OS usage

#### 7.3.1 Scheduling

This chapter explains important aspects of task scheduling for projects that are derived from platforms. Such projects are exposed to a risk not to meet their run-time constraints. Special care should be taken to configure the system properly. The topics below should be taken into account by SW Architects, Integrators, Developers and Team Leaders. It is important to understand that there is no perfect receipt to mitigate the issues. Every project has own specifics that may require actions, different than the described hereafter.

The point is that project must have systematic approach when dealing with CPU load and task scheduling.

The starting point for projects is to realize that this is complex hybrid system running on one and the same CPU. There are two main parts, having different nature:

- Primary Vehicle and Safety System part, called below PS:
  - The expectation here is to have real-time behavior.
  - It is based on non-preemptive periodic task execution with low jittering.
  - Fast ISR execution.
  - Everything must be predictable and to fit the defined deadlines.
  - Usual the selected task policy is Rate-Monotonic (see link below).
- Secondary graphical system, called below GS:
  - The expectation here is to have fast reaction to user events and smooth animations.
  - It is fully event driven system, typically with high jittering of the tasks.
  - Several preemptive tasks are used.
  - Complex management of graphical HW.
  - Not predictable in terms of reaction time.
  - Performance drastically depends on the provided HMI model.

Obviously the same rules for making the task scheduling cannot be applied for both. The approach that is used to set CPU quotas for each part.

#### Setting CPU quotas for PS and GS

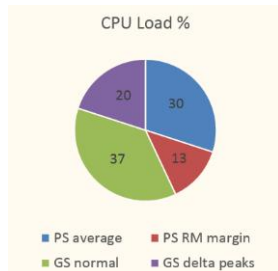
Based on the measurements of the existing projects it is observed that usually PS takes 30...40 % CPU load, and GS takes 60...70 % (xJCI statistics). Normally, there is a lot of job to be done by HMI related tasks, so, it is fair enough GS to receive bigger quota. But how much exactly? Because the PS CPU load is more stable (not so much peaks at run-time compared to GS systems) we can start our speculation based on it. Imagine that PS takes 35 % average CPU load. Taking into account that *Rate-monotonic* is reliable to ensure deadlines of the tasks when the average CPU load is  $< 70\%$ , we have  $35 / 0.7 = 50\%$ . This means that if we give 50 % CPU to PS it can be properly working under rate-monotonic schema. And we have 50 % left for GS → but this might not be enough.

Balancing the CPU load in a particular system is iterative. We may set first limit for PS to 30 % for example, because this is the minimum from the range above. The quota can be adjusted when new functionalities are added. It will depends on which side they are added (PS/GS).

For GS we have 70 % in that case. These cannot be assumed as average load. They represent the limit that should never be exceeded by GS, even when it is flooded with user events (like button clicks) and needs to draw a lot. Here the point is to create test scenarios for particular HMI model, which represent use cases for nominal usage of the GS with normal event flow with drawing the most complex scenes/transitions/animations. With the measurement results from these test we will know what the average CPU load of GS under normal operation is. Then we need to know the peak CPU load. It might happen in several scenarios - we need to take the most consuming one. After we need to look at the average and peak load of GS:

- If they are too close, we have less issues. But this is rarely happen in event driven systems.
- If the gap is huge, there is a problem in the design of GS. Most probably parts of it are overspending something - a design review needed to find the issue.

The starting point could be to say that 20 % overhead in case of peak load, compared to the average load, is acceptable. Thus we have:



, where "RM margin" means rate-monotonic margin.

The numbers in the pies are elaborated this way:

- Assumption: PS average load takes 30 %. This must be 70 % from its quota to guarantee rate-monotonic is schedulable. I.e.

$$\text{PS Quota} = 30 / 0.7 = 42.857... \sim 43 \%$$

- PS RM Margin =  $43 - 30 = 13 \%$
- GS Quota =  $100 - 43 = 57 \%$
- Assumption 20 % peak load for GS means that:

$$\text{GS Normal} = 57 - 20 = 37 \%$$

So, we reached an initial split of CPU quotas of 57% GS and 43% PS.

Once these initial quotas are set the game is to design and/or optimize PS and GS to fit. This is a constant care that must be taken. If we see that the initial assumption is not OK, then we need to adjust. At the moment when 100 % functionality is coded it is expected that adjustments are no more needed.

The trap that must be avoided during the adjustment process is to avoid shifting CPU quotas PS to GS, GS to PS, PS to GS ... Analysis of what is consuming the time needs to be performed on every step and the guilty part to be optimized/redesigned. Not just to extend its quota.

Also it is important to ask for expert opinion if there are indications that may be the power of the chosen CPU is not enough anymore to handle the new functionality. This must be done as early as possible.

#### Task priorities PS

They are usually set according to rate-monotonic approach. See the links below:

[Rate-monotonic scheduling](#)

[Earliest deadline first scheduling](#)

Possible areas for CPU load reduction:

- Remove all useless tasks and ISRs - even empty they are spending some CPU time.  
Example: Timers can be used without ISR defined.
- Increase the periods of the tasks, if it is not needed to calculate so frequently.
- Reduce the duration of ISRs (CAN, ADC, Sound Driver, Stepper Motors, Raster LCD Drivers are typical cases). Only perform necessary actions within ISR (e.g., storing of peripheral data registers). Algorithms/logic should not be included within HW ISRs.
- Check the possibility to use DMA instead of having frequent ISRs calls - Example: SPI ISR on every received byte.
- Reduce the duration of the periodic tasks:
  - Direct code refactoring - less instructions to be used:

- Remove massive or extra protections in static functions. It is enough that input parameters are protected when public functions are called. Then all static functions work in a "protected" environment.
- Protections inside inner loops that could be moved to outer loops or to be called just once.
- Frequent index operation [] in the same function or in a chain of functions to be replaced with variable in stack (pointer or alias in C++). Only usage of this variable. This is proven that is more effective with or without compiler optimizations.
  - Things that can be done on event like Init, Goto Sleep, Wakeup etc... to be moved outside periodic tasks.
  - Replace algorithms with more effective, especially seeking in large data storages or complex calculations.
  - Use mathematical approach to do equivalent calculations in an optimal way.
- Duplicated job over the time to be done only once and other places to take the result that is already cached.
- The duration of all kinds of locking should be decreased:
  - Interrupt EI/DI, Suspend/Resume
  - Lock/release OSEK resources
  - Critical sections, Mutexes
  - Discover algorithms that do not require any locking! In some cases this is possible and improves significantly the performance.

Because these could be applied to every task, where to start from?

- The most frequent interrupts. Usually these are communication ISRs (CAN, SPI, UART).
- The most frequent periodic and aperiodic tasks.
- The longer non-preemptive tasks that tend to become preemptive soon (HMI, Gateways).
- Segment and raster displays (rendering or texts and images) are usual candidates for code refactoring. Huge gain with concentrated fixes in small number of places.
- Libraries for common use - they have drastical impact to the whole system. Example: 2..3% average CPU load can be saved just by optimizing memset/memcopy or more effective string operations.
- Make a dedicated review of all communication mechanisms - CAN, IPC, Video buffers ... You may find that there is useless data copying.
- Is there any run-time decompression that can be avoided?
- Redundant or slower encryption/decryption?

#### Task priorities GS

The priority order of GS's tasks should be as follow:

- (lowest) DP Connector (gateway) task - transfers data from RTE to HMI and backward
- Main GUI application task
- Timer management
- Datapool processing
- All renderer tasks - they must be here to ensure needed FPS for animations

Except all abovementioned optimization hints one can look at these:

- Consider use of xJCI fonts instead of TTF (Flash for CPU)
- Consider EBGUIDE model optimization:
  - If you have complex views with many widgets, consider creation of custom widget which implements the EB model logic for this view.
  - Avoid use of PropertyVector and ListElementReference in the model
  - Avoid assignment array = array in the EB guide model
  - Avoid assignment string=string in EB guide model. Initialize Text.textContentTable with multiple texts and just change Text.selectedTextContent.
  - If you have animation over container with many widgets, consider use of pre-renderer for this container.
  - If you use TTF fonts consider use of pre-renderers for TextWidgets (RAM for CPU)
  - Decrease the count of the widgets in views for which the CPU load is high



- Try to decrease the count of the calculations and initializations inside the EB guide code (state-machine, actions etc). If you have complex initializations and calculation consider moving them outside in the Gateway component.
- If you don't use TTF fonts enable RENDER\_OPTIM and triple buffering of GEN2. This will increase the FPS, but could reduce the CPU load.
- Consider implementation of adaptive system which enable or disable dirty-area-optimization of GEN2. "Dirty are optimization" is effective when there aren't many changes on the screen, when almost everything changes, it become very ineffective.
- When you write many data items to HMI datapool, consider explicit locking of data pool until all data is written.
- When you have multiple events (button events for example) in Gateway component, consider locking of data pool and sending them to the HMI at once. This will reduce the count of the needed redraws.
- Consider change of the TTF engine (IType vs FT2, provides different results in different situations).

#### Use tools to find the right scheduling schema

In general, the optimal scheduling schema for a particular system can be obtained with a tool, which solves optimization problems. There are commercial tools doing this. They use the parameters of the tasks and interrupts (capacity, deadline, max latency, periodicity, priority) and based on a selected scheduling policy an acceptable solution is provided. There is internal command line tool named *scheduler* which is doing a simulation for a certain seconds after system is started. It could help in analysis to determine what needs to be changed. Please, contact Ilian Penchev (ipenchev@visteon.com) for details.

#### Technical Limits

Please, do not forget that CPU load optimization and respective task scheduling policy of one system is not for free. It is a constant trade-off between CPU load, RAM and ROM occupation. SW architects/design champions must have clear concept what to sacrifice in favor of what.

### 7.3.2 Using OS tasks as containers with sub-tasks, activated by prescalers

The goal of this article is to explain the consequences of using prescalers within periodic OS tasks. Nowadays several projects are using this method for different reasons, but there are issues leading to the need for source code modifications.

Here are some of the reasons leading to usage of prescalers within periodic OS tasks:

- OS cannot be configured to handle the needed amount of periodic tasks.
- There is strong relation between periodicity of some tasks, so one decides to keep them always together.
- Some tasks need periodicity that is incompatible with the main scheduling schema, defined at system level.

Mainly two techniques are used to achieve this:

1. Use internal counter in the component, which is initialized to zero at system startup/wake up and then incremented by one each time a dedicated periodic runnable of the component (the task) is called by the OS. Modulo operation (%) is used with different prescaler values. When it is zero the respective sub-tasks are called.
2. Periodic runnable of the component is using timer services to measure how much time has elapsed the previous call. Different sub-tasks are activated when certain timer is elapsed. Then timers are restarted.

The second one spends more ROM for code and more CPU to call the external timer services, but is more robust. This is because it uses timers that are sourced by HW counters (roll-over 32-bit registers usually). In case of overloaded system and/or significant jittering of the OS tasks the deltas between HW register values are still adequate, no matter that they are read later than expected. Even it is possible OS task not to be called at all for some time.

No matter what is the chosen implementation there are drawbacks when prescalers are used:

#### The sub-tasks are strongly bound together

- When their OS task container is moved by SW integrator to another frequency, all of them are moved at once.
  - In most cases nobody verifies whether all sub-tasks are still in their operational range of periodicity! This is risk of regressions.
  - Each sub-task, based on its detailed design can operate properly within a certain frequency range. If it is put below the minimal frequency the code will overspend CPU to calculate several times the same inputs. If it is above the maximum we will have erroneous calculation and wrong outputs. It is difficult for SW integrator to find the optimum - he/she needs to dive in the implementation of the prescalers to understand where the borders are.
- Balancing the scheduling in some cases will require setting an offset for activation of one or another sub-task. This is not possible, because the offset can be set only to an OS task. Sub-tasks are completely invisible to the OS.
- Imagine that one of the sub-tasks is eating too much CPU. SW integrator would like to move it to another slot with the same frequency, which is shifted from the original one with some [ms]. Again this is not possible, because he needs to break the encapsulation of the component to access the sub-task.
- Also the whole container is either preemptive or non-preemptive. Not possible to have some of the sub-task preemptive and the others not.
- There are standard systems for CPU profiling - external and internal to the company. They are based on the OS tasks. Special extensions need to be implemented to instrument code for measuring every single sub-task. Unfortunately this is done today in different way in different projects. Typical cases are Autosar systems where several runnables are inside one OSEK task. The interest is to measure every runnable in order to identify the respective component that needs to be optimized for performance.

#### Implementation of prescalers in one place

Usually this is the body of the OS task. The implementer must pay attention to the next points:

- Overhead of prescaler calculation must be minimized:
  - Usage of modulo (%) operator and if/elseif constructions is spending more CPU than needed. Replace them with equivalent masks and ordered indexes in switch/case - this allows compiler to optimize them in tables with offsets.
  - Pre-calculated scheduling tables in ROM are effective - use them where appropriate.
- Every modification in such code must be carefully reviewed/tested, because it might affect another sub-task which periodicity is not intended to be changed.
- Parameterization of the functionality is difficult and could lead to additional CPU loss. Imagine that some of the sub-tasks must operate under specific conditions (NVM or run-time ones). This means that additional logic should be mixed with the prescaler one. Normally, such things are implemented at system level with the means of OS or RTE which are using different task containers for different modes/configurations. So, we will have duplication of the work in the component that uses prescalers - just to do the same.

#### Optimization limitation

The most important drawback is the fact that coexistence of several components, behaving as schedulers, leads to difficulties or impossibility to use tools for CPU load optimization.

The point is that one needs to describe the tasks with their periodicity and capacity (average execution time, usually taken from SW integration reports) and deadline. Obviously the first two are different for every sub-task within the OS task container, and we must provide only one set of parameters. Even if separate sub-tasks are measured and parameters are given to the optimization tool, the optimal result will be based on the assumption that these sub-tasks can be tuned

individually (by slots, offsets, chaining etc...). This cannot be applied unless the implementation of the prescalers is removed/changed.

#### Recommendations

- Avoid creation of own schedulers!
- Use as much as possible the capabilities of OS to configure task scheduling - like offsets, scheduling tables, non-preemption groups etc...
- SW Architect is responsible to define the right scheduling policy for the application and to put constraints to SW components about how they should behave.
- SW Integrators must obey the rules for the policy and to inform the SW Architect if the system experiences issues (measurements during the integration process). In such case a global solution must be applied, instead of doing local fixes at one or another place to patch the current defect only. Root cause must be known and then the fixed scheduling policy (or some of its rules) should be deployed to the whole system.

### 7.3.3 Demo SW scheduling

#### 7.3.3.1 Mapped runnables to tasks proposal:

Task	Runnable	Period+offset	Prio/Type	Comments
OsTask_10msCAN	Can_MainFunction_Read(); Can_MainFunction_Write(); CanSM_MainFunction(); ComM_MainFunction_0(); Com_MainFunctionRx(); Com_MainFunctionTx(); Nm_MainFunction(); CHmiCtrl_Impl_MainFunction();	10+1	200 NON	
OsTask_10msEcuM	CEcumExt_Impl_MainFunction(); EcuM_MainFunction(); BswM_MainFunction(); Dlt_MainFunction(); Fee_MainFunction(); NmM_MainFunction(); Fls_MainFunction(); Ea_MainFunction(); Eep_30_XXi2c01_MainFunction(); I2c_MainFunction(); TmExt_MainFunction(); CloHwAb_Impl_MainFunction();	10+2	200 NON	
OsTask_Gdt	CGdtCdd_Impl_MainFunction(); GdtCtrlRef_SpeedDataRx(); GdtCtrlRef_TachoDataRx(); CGdtCtrl_Impl_MainFunction();	16	190 NON	Currently CGdtCtrl_Impl_MainFunction is doing nothing
OsTask_20msHP	Com_ReceiveSignal(ComConf_ComSignal_TachoRef CEngMdl_Impl_MainFunction(); CSpdMdl_Impl_MainFunction();	20+3	200 NON	
OsTask_20msMP	CStpCdd_Impl_MainFunction(); CPtrSpdView_Impl_MainFunction(); CPtrTachoView_Impl_MainFunction(); CPtrEctView_Impl_MainFunction(); CPtrFuelView_Impl_MainFunction(); CWrrnCtrl_Impl_MainFunction();	20+4	180 NON	
OsTask_30ms	CBtnMdl_Impl_MainFunction CTtMdl_Impl_MainFunction CTtView_Impl_MainFunction CanNm_MainFunction	30+5	150 NON	

## Software Architecture Specification

OsTask_30msSnd	CSndCdd_Impl_MainFunction CSndView_Impl_MainFunction	30+6	150 NON	
OsTask_50ms	CModMdl_Impl_MainFunction CTripMdl_Impl_MainFunction CodoMdl_Impl_MainFunction	50	100 NON	
OsTask_100ms	CDimMdl_Impl_MainFunction(); Can_MainFunction_BusOff(); CDimView_Impl_MainFunction(); Can_MainFunction_Mode(); CEctMdl_Impl_MainFunction(); CFuelMdl_Impl_MainFunction(); Can_MainFunction_Wakeup();	100	50 NON	
OsTask_Render			4/FULL	
OsTask_StackMeasure	CDD_Test_StackMeasure		2/FULL	Optional
StartupTwo				Call only once
OsTask_2ms	CSndCdd_Impl_ToneProcess		250/NON	
AppTasksLP_10ms	CDD_Test_MainFunction		10/NON	Should be removed
OSTask_FOTA	CFotaCtrl_Impl_MainFunction CDD_FlashCdd_MainFunction		3/FULL	Lowest prio pre-emptive task. Period = 8 ms, 1000 activations
OS_IdleTask			1	

### 7.3.3.2 Recommendations

**Assumption is taken for the optimization proposal that each non pre-emptive task is taking time lower than 700 ... 800 us!**

The following things should be done inside the build:

1. Periodic iohwab reading should be split in 2 -> ai and di reading
2. Adc reading could be done continuous without periodic start
3. Stack measurements should be done inside pre-emptive task because takes more than 200 ms.
4. Schedule table with offsets should be used for the 10ms, 20ms and 30 ms tasks
5. Stack calculation to be in a separate pre-emptive task.
6. RenderTask activation should be generated through the rte with Runnable calls. (This is more to be prepared for the future safety strategy)

### 7.3.3.3 Cpu load tool observations

The following observations should be taken into account when the Cpu load tool is used. This tool add its own deviation(load) of the real behaviour

For 80 MHz clock the following measurements are done:

- Cat2 ISRs are adding ~18 us overhead;
  - Pre ISR and task hooks are taking ~20 us;
  - Post ISR and task hooks are taking ~20 us;
  - If Cat1 ISR is used manual adding of hooks should be done if measurement is requested.
- Inside the hooks protection with \_\_DI()/\_\_EI() should be done

Too frequent ISRs will increase the max time of the task where the ISR happens.

Stack measurement should not be done runtime, but a dump of stack should be provided and external program should analyse it and calculate the usage.

### 7.3.4 Rationale for FotaCtrl

FotaCtrl should run on the lowest priority as it should not affect the system. As it as to poll the Flash for ready ness it should be allowed to continue to run as soon as possible.

During Decompression a huge amount of data can be brocessed which can take longer than the up to 800us which a normal task should run. Also allowing to run longer allows bigger buffer to be processed without restructuring the code to do atomic operations, which costs memory and performance.

Also bigger buffers allow more efficient compression.

Finally FotaCtrl will perform a SHA2 versification and RSA operation. These operations take 400ms for reasonable block sizes so can't be implemented in a typical task, too ,if they are not drastically restructured.

The Mainfunction of the FlashCdd should run in the same thread, as it has to poll for the flash to be ready or not – not disturb others and to do this only when the system is otherwise idle it should be done there. Obviously this is a drawback that FlashCdd and FotaCtrl can't operate at the same time – but as we don't have buffers Fota has to wait anyway for the FlashCdd to be come ready first, before further data can be produced.

## 8 Functional breakdown

### 8.1 COM infrastructure

#### 8.1.1 Requirements

This chapter contains global requirements for COM infrastructure. Will be moved to requirements tool, when decided.

COM01	<b>Minimum signal states for signal</b>
Type	Requirement
Priority	High
Description	Each com signals shall have at least state Init, Present, Absent with configurable timeout
Exceptions	Allowed with agreement of Design Champion/SW Architect.
Comments	Partially accept the proposal, with further clarification about minimum set of states for signal qualifier. Limit to number of predefined number of states.

COM02	<b>Possibility to add signal states for signal</b>
Type	Requirement
Priority	High
Description	The framework shall allow to add additional state like "Default", "New", "ConfirmedAbsent" or "Safety" for individual signals
Exceptions	Allowed with agreement of Design Champion/SW Architect.
Comments	New shall be "optional" has it consumes a lot of memory (One per consumer)

COM03	<b>Notification on signal transmission completion</b>
Type	Requirement
Priority	High
Description	It shall be possible to be notified when a signal is successfully sent on the network or not (Timeout)
Exceptions	Allowed with agreement of Design Champion/SW Architect.
Comments	PSA request to count 3 sending for event. (managed by application) Note: to provide feedback at physical transmission completion

COM04	<b>Possibility to reuse Client/Server</b>
Type	Design constraint
Priority	Low
Description	It shall be possible to send data by C/S as Receiving
Exceptions	Allowed with agreement of Design Champion/SW Architect.

Comments	Mapping data per data is very long. It is better to link directly some port-interface to save time. Assumption is that IPC communication is reusing multi-core communication.
----------	--

COM05	<b>Factorize common parts in communication stacks</b>
Type	Requirement
Priority	High
Description	To factorize the common part between communication stacks within a NAS Architecture.
Exceptions	Allowed with agreement of Design Champion/SW Architect.
Comments	Factorization for all the stack, comparison matrix to be developed SOSA / STK @Etienne BSW_NAS

COM06	<b>Implement COM stack for AS and NAS architecture</b>
Type	Requirement
Priority	High
Description	Taking into account the possibility to implement a COM stack for both AS & NAS architecture. For example: such component / framework would manage the frame encoding / decoding.
Exceptions	Allowed with agreement of Design Champion/SW Architect.
Comments	If COM stack for implementation is derived from OSEK COM requirements, it's accepted, otherwise out of scope. See #11, @Etienne BSW_NAS

COM07	<b>Provision generation of signal data interface for RTE</b>
Type	Requirement
Priority	Mandatory
Description	provision of signal oriented data interface for the RTE
Exceptions	Allowed with agreement of Design Champion/SW Architect.
Comments	generated interfaces for data access

COM08	<b>Enable start/stop of messages transmission</b>
Type	Requirement
Priority	Mandatory
Description	communication transmission control (start/stop of I-PDU groups)
Exceptions	Allowed with agreement of Design Champion/SW Architect.
Comments	start stop sending and receiving messages

COM09	<b>VFB signals communication</b>
Type	Requirement
Priority	Mandatory
Description	sending of signals according to transmission type as specified in the VFB specification
Exceptions	Allowed with agreement of Design Champion/SW Architect.
Comments	event or periodical transmission, enable/disable for periodical messages

COM10	<b>Support different notification mechanisms</b>
Type	Requirement
Priority	Mandatory
Description	different notification mechanisms (event, functional call, signals, callbacks, callouts)
Exceptions	Allowed with agreement of Design Champion/SW Architect.
Comments	signal, functional, events, callbacks on communication  Handling of timeouts, confirmation feedback

COM11	<b>Support integration of E2E library</b>
Type	Requirement
Priority	Low
Description	Use of E2E framework
Exceptions	Allowed with agreement of Design Champion/SW Architect.
Comments	Look for BSW and E2E integration

## 8.2 Warnings management

After discussion between SW Architects xJCI warning system is accepted as a basic warning management concept. Almost the same concept is on Visteon Classic side.

The next points were accepted:

- No road blockers to use GDT UI messaging system.
- Agreed requirements/decisions:
  - Number of vertical channels, managed by the warning system must be parameter, i.e. channels may vary.
  - Centralized warnings management will be applied, but every channel (sound, telltales, HMI etc...) can manage additional specifics like priorities for ex.
  - WrmCtrl is acceptable to work on polling for now.
  - Remapping of Warnings IDs for HMI systems to be investigated and finalized (GDT, Segment, ALTIA ...)
  - Detailed design of warning system must be improved.
  - Parameterization for min/max indication timeouts, prioritization channels and dependencies to be considered.
  - Specific channel for add image / icon for graphics indication to be supported.

**Note: This section will be extended with full description of the warnings management concept!**

Warnings are defined differently than customers define them!

Usually customer says that warning is something that appear on the screen as a text. The extended definition is:

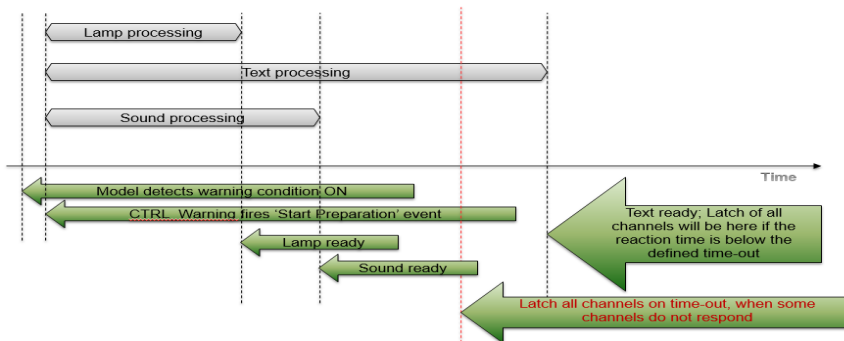
*A warning is a notification to the driver in the car, which consist of any combination of indicators that have to appear simultaneously.*

**Anything that is able to be managed with the decided features of the warnings should be treated as warning. Even OEM does not defined it explicitly.**

In other words, output indicators like Telltales, LCD segments, Sound etc. can appear in any combination. Every particular combination of those should be managed as a warning. Even a separate LED (a telltale) is a warning.

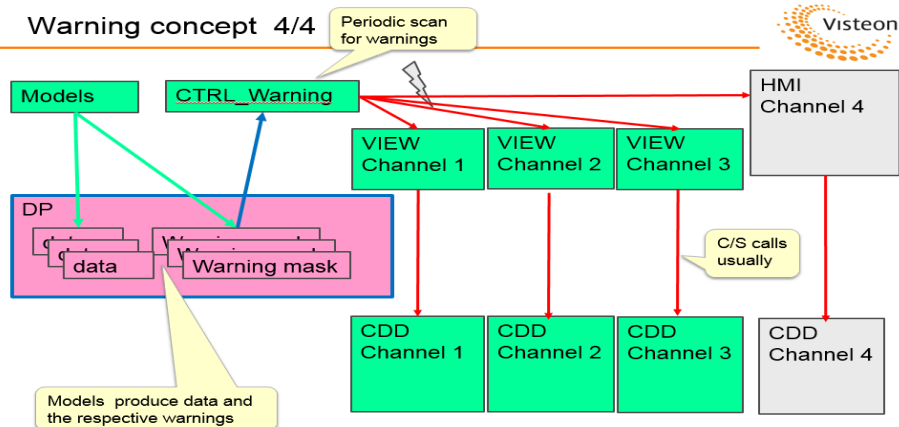
If we take some particular case:

- Lamps 1st driver's indicator channel; all lamps with unique IDs
- Text on display 2nd driver's indicator channel; all texts with unique IDs
- Sound 3rd driver's indicator channel; all sounds with unique IDs



Processing one warning, consisting of 3 driver's indicator channels.

#### Warning concept 4/4



### 8.3 Watchdog management

#### 8.3.1 TA

The Wdg management in TA is done by the standard AS BSW CMPs called WdgM, WdgIf and Wdg. Fig 1 inside [R18] is showing the interactions of the Wdg related CMPs inside the AS stack. The purpose of the WdgIf is to be able to map different Wdg-s.

WdgM and WdgIf are implemented by TTTech Automotive GmbH and integrated by Vector. They are integrated inside the SIP delivery(<delivery id="CBD1400814\_D01">). Detailed description on what is implemented inside could be found in [R17] and [R18].

[R11] is giving details regarding the functionalities and details on what should be done inside. The standard introduce 3 types of monitoring:

- *Alive Supervision* (see Chapter 7.1.5)
- *Deadline Supervision* (see Chapter 7.1.6)

Copies of this document are uncontrolled



- *Logical* Supervision (see Chapter 7.1.7)

According to [R18] all the 3 types of monitoring are implemented inside. Systems with pre-emptive tasks identified as ASIL A and B usually require *Deadline* monitoring. *Logical* supervision is necessary only for ASIL C and D, which are usually not used in the targeted domains.

If there is OS SC3, the Scheduler might need such monitoring (*Logical*).

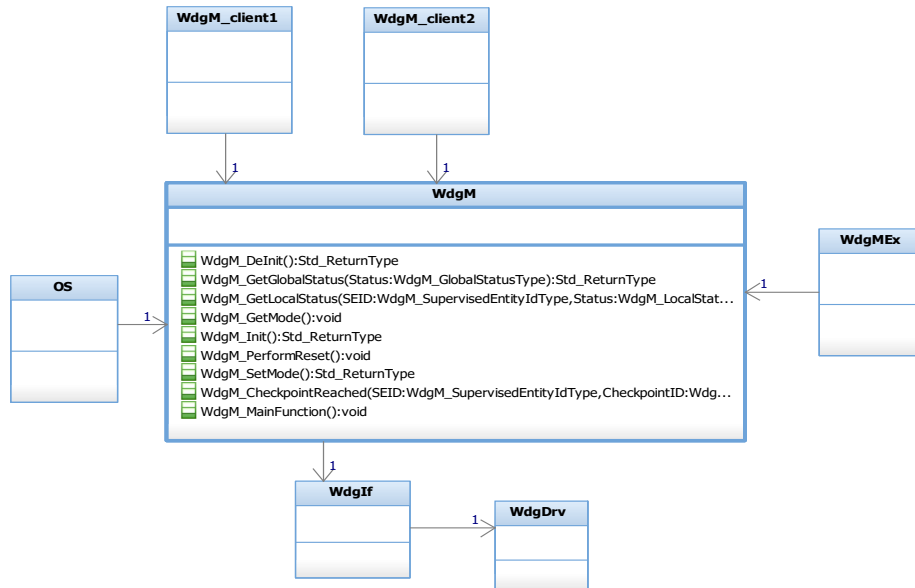
In AS 4.2.1 the Wdg is cleared inside ISR. There are 2 GPT ISRs inside the driver:

- GptChannelConfiguration\_Wdg\_Trigger – used to clear periodically the watch dog.
- GptChannelConfiguration\_Wdg\_TO – used to act as watch dog counter for the SW. Its counter is cleared inside the WdgM\_MainFunction(). If not cleared on time the notification of the GptChannelConfiguration\_Wdg\_Trigger is disabled and a watch dog reset will occur.

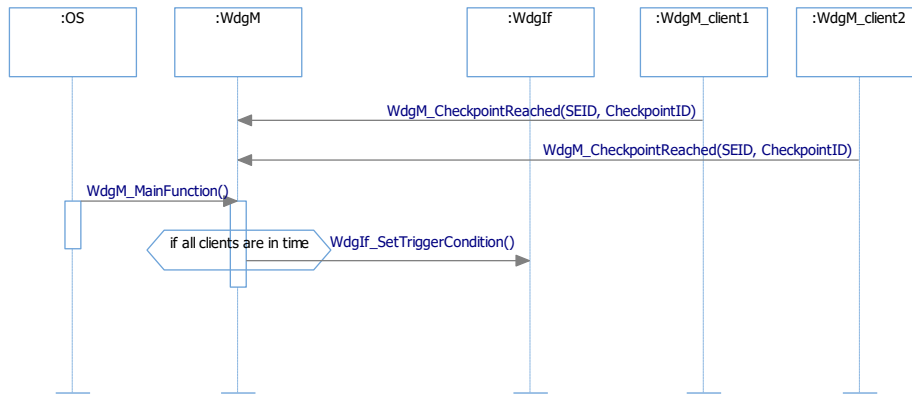
### 8.3.2 TB

For the needs of TB, the component WdgM should be created. Requirements on how this to be done could be found in [R11]. This component should be able to maintain *Alive* and *Deadline* supervision. Deadline supervision is available in Gen2 - component SRV\_GSS. Its functionality could be used to satisfy the SRS [R11].

Interfaces which should exist inside could be seen in the diagram below:



Principal operation of the WdgM (*Alive* supervision), once initialized and configured is shown on the sequence diagram below:



### 8.3.3 WdgM integration

WdgM is already integrated by Vector inside the SIP for TA. Details about the integration requirements could be found inside [R18] chapter "Integration". Inside the document cyclic call at 10 ms for the WdgM\_MainFunction() is proposed. The priority of the task should be as high as possible.

### 8.3.4 WdgM configuration

The strategy for configuration and monitoring of the SW will be updated as soon as the ASIL concept is done. Meanwhile a monitoring of the lowest prio non pre-emptive task should be done. This way normal periodic call of all non pre-emptive tasks will be ensured. For the pre-emptive task a Deadline motiring should be configured and used. Points where such monitoring could be applied should be additionally specified by the GDT team. To configure correctly the WdgM a statistics should be gathered. This could be done in the following ways:

- CPU load analysis;
- Disabling the WdgM reset and applying a stress test to the system. When finish the max values of the counters should be taken and used as configuration values.

If the values are too big additional analysis should be done and the system should be optimized!

## 8.4 NVM management

### 8.4.1 TA

NvMExt is removed from the architecture. TA & TB are using AS 4.2.1 compatible RTE which manages Nvm data elements.

Scenario #	1	2	3	4
Name	Gen 2 Like	Gen 2 + handles	Gen 2 hybrid	Autosar 4.2.1 Compatible
Description	C/S interfaces Uses individual RTE ports for data items	C/S interfaces Uses unified port for all data items	C/S and S/R interfaces, configurable per data items.	Use native Autosar 4.2.1 approach to have buffers in RTE and all in configured by DaVinci.

		Generated macros and handles for R/W access, NOT published in RTE		
Pros	Smallest effort to make it.  All knowledge in place.  Simple TB RTE generator.	Minimized ports in RTE. Component modifications w/o DaVinci licenses for config.  Simple TB RTE generator.	RSA/Nissan require S/R - OK  C/S with or without handles could be used by some projects.	We might be able to remove NvMExt.
Cons	RSA/Nissan require S/R.  Require non-preemptive callers.	RSA/Nissan require S/R. Special treatment of this case is possible.  Require non-preemptive callers.	More complex NvMExt.  More complex TB RTE generator.	First time to use this. Lack of people/knowledge to maintain TA tools. AS 4.2.1 meta-model affects UML & Instancer, OBC and RTE ARXML export.  Complex TB RTE Generator needed.
Conclusion	No improvement since Gen 2 à this is not platform goal.	Suitable for temporary solution for 1.3.	More like a patch than a solution. Too much effort for a temporary patch for 1.3.	Long term desire, definitely not feasible

<b>Feature</b>	<b>Current capability - SRV_Cfg</b>	<b>ASR 4.2.1</b>	<b>Analysis</b>
Management of R/W of individual data elements	Supported via C/S	AUTOSAR meta-model extended to support NvDataInterface	Future of AUTOSAR is to use S/R and controller interface Renault/Nissan uses S/R C/S needs to be protected by multi-core and MPU If TA 1.2 solution used, we need to investigate the protection mechanisms <b>Conclusion: We need to migrate to S/R</b>
ROM defaults to be supported (in case of failure)	Supported	Supported	No significant differences

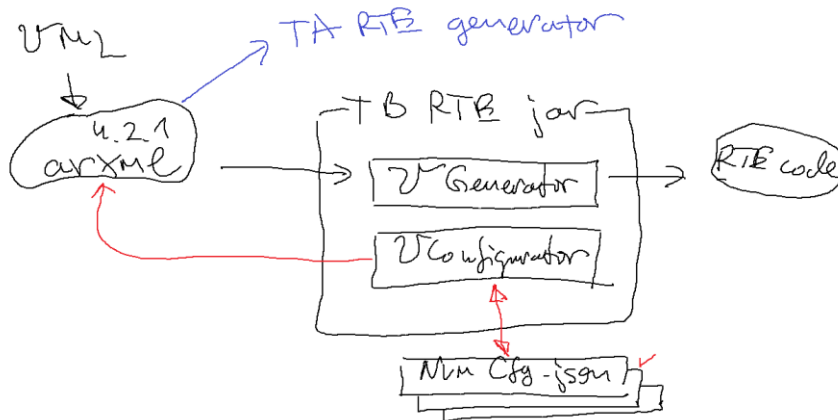
Callbacks on end-of-write of individual NVM blocks	Supported - notifies a set of users when block write ends	TBD	CDD are tightly coupled to NvMExt <b>Conclusion: Callbacks need to be reversed</b> Impacts StpCdd from TA 1.2
Callbacks on end-of-read of individual NVM blocks	Supported - notifies a set of users when block read ends	TBD	CDD are tightly coupled to NvMExt <b>Conclusion: Callbacks need to be reversed</b> Impacts StpCdd from TA 1.2
Immediate and Delayed write block	Not supported	Supported	<b>Conclusion: This should be added to Turing solution. This will allow callbacks to be removed.</b> <b>Conclusion: Parameter will be applied to the block</b>
Check of same-data write	Supported	Not supported	<b>Conclusion: We need to remove from Turing so we can migrate away from NvMExt in the future.</b>
Odometer data sets (wear out protection) - ONLY APPLICABLE TO EXTERNAL EEPROM	Supported	Not supported	<b>Conclusion: This functionality should reside in the component (OdoMdl) or adapter component</b>
Interpolation via handles (used by Stp)	Supported	Not supported	<b>Conclusion: This functionality should be moved to requesting component or as extension to a library. This is high level logic above NvM.</b>
Array access functionality	Supported - individual array elements accessible	Supported - access full array	<b>Conclusion: Utilize ASR functionality</b>
R/W by memory address (used by manufacturing)	Supported	Not supported (and FORBIDDEN)	<b>Conclusion: We need to wait for Diag and MEET-SW design before implementing this. This is out-of-scope for TA 1.3</b>
Support for two providers of same data	Supported - via C/S interfaces	TBD	<b>Conclusion: This is an exception case and is out-of-scope for TA 1.3. If necessary, a new provider will be created</b>

			to perform the arbitration.
Reprogramming	Supported - via Diag services, easy due to sequential	Supported - but difficult due to creation of structures	<b>Conclusion: This is out-of-scope for TA 1.3 because we do not support Diag. We have work-around for TA 1.3 using debugger to make the block "dirty".</b>
Encryption	Supported - via encryption library	TBD - CRY/CSM?	<b>Conclusion: This is out-of-scope for TA 1.3</b>

Conclusion: NvMExt can be removed. ASR 4.2.1 solution is applicable. TB RTE generator will become more complex.

Result: Configuration of Nvm will be simpler. Effort needed for RTE generator design will increase, but effort for NvMExt is eliminated.

TBD: Determine if we can avoid usage of DaVinci



Inside [R13] fig1 is given an overview of the memory HW abstraction. This abstraction is provided by the following AS components:

- Fee - for flash memory – detailed description could be found inside [R14] and [R16]
- Ea - for EEPROM - detailed description could be found inside [R13] and [R15]

The abstraction is necessary to the upper layers to be independent from the device specific addressing scheme and segmentation and provide the upper layers with a virtual addressing scheme and segmentation as well as a "virtually" unlimited number of erase cycles. More details on addressing scheme and segmentation could be found inside [R13].

In Turing demo only Fee is used because data flash is used to store the NVM data. The Fee use FIs to read/write data from flash.

Configuration of the Fee is done inside DaVinci configurator. The blocks configured inside could be seen in the table taken from Turing 1.2 demo and shown below:

FeeBlockConfigurations	Block Id	Block Number	Block Size [Byt]	Immediate Data	Number Of Datasets	Number Of Write Cycles
FeeClusterSettings	0	0x2	16	<input checked="" type="checkbox"/>	2	10000
FeeConfigBlock	2	0x4	4	<input type="checkbox"/>	2	10000
FeeInterpolations	3	0x8	40	<input checked="" type="checkbox"/>	1	10000
FeeRuntime	1	0x6	8	<input checked="" type="checkbox"/>	2	10000

More details on the configuration could be found inside [R16] and inside Turing\_Fee\_ecuc.arxml.

#### 8.4.2 TB

TBD

### 8.5 Diagnostics

#### 8.5.1 TA

The diagnostics management in TA is done by the standard AS BSW CMPs called Dcm. More details could be found in [R19] and [R20].

Dcm should be able to receive and transmit diagnostic can messages from Can, CanTp and PduR.

DcmExt component is introduced, who will handle data conversion from the diagnostic request buffer to the Swc API provided by the component.

Dcm functionality should be used as much as possible to avoid DID, service ID and length checks on more than one place.

##### 8.5.1.1 Design alternatives

###### 8.5.1.1.1 Connection between Dcm, DcmExt and Swc/Cdd

A1: There is idea to use a standard interface (TI\_Diag) as connection between the Swc/Cdd and DcmExt. This connection will pass through RTE. It will minimize the connections inside the RTE. Inside the Swc/Cdd a file xxxDiag.c will own the runnable DiagAction and will do internally calls to the Swc/Cdd functions.

A2: To use direct connection between Dcm and Swc/Cdd. In that case runnable will exist for each connection inside the Swc/Cdd. The standardized TI\_Diag is not going to be used or will exist in parallel. This will increase the connections inside the RTE.

###### 8.5.1.1.2 The following strategies could be applied for the DcmExt implementation:

B1: To do the configuration as much as possible inside the Dcm. Callback functions to have implementation inside DcmExt.

B2: To set callbacks inside the Dcm for the following service handlers before DID check:

- 0x11 EcuReset handlers
- 0x22 ReadDataByIdentifier handlers
- 0x23 ReadMemoryByAddress handlers
- 0x2E WriteDataByIdentifier handlers
- 0x2F InputOutputControlByIdentifier handlers
- 0x31 RoutineControl handlers
- 0x3D WriteMemoryByAddress handlers

This mean all the length and DiD checks will be implemented manually inside the DcmExt.

Advantages:

- Each extension of the diagnostic with new DID it will be manually implemented and no Davinchi configurator will be used;
- TB will be able to reuse it.

Disadvantages:

- The big disadvantage is that there will be a lot of code inside the Dcm not used;
- Manual implementation.

##### 8.5.1.2 Design choices

Decision was taken to apply choices A1 and B1.

#### 8.5.2 TB

Copies of this document are uncontrolled

Page 78 of 119

TBD

## 8.6 DTCs management

### 8.6.1 TA

DEM in TA is done by the standard AS BSW CMPs called Dem. More details could be found in [R21] and [R22].

### 8.6.2 TB

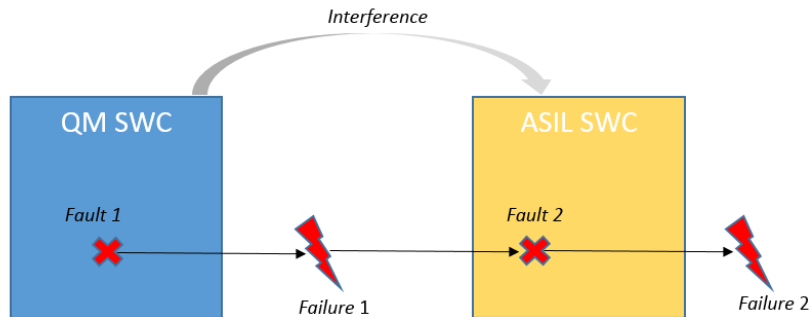
TBD

## 8.7 Safety

This chapter is supposed to provide principles for designing safety relevant system. This does not mean that the goal is to make the entire SW "safe", but only a part of it. The targeted Automotive Safety Integrity Level (ASIL) is ASIL B.

Analysis shall start with feared events or safety goals that are defined for a given project. Based on this analysis the SW shall be structured into 2 parts: Quality Managed (QM) part and ASIL part. Normally the ASIL part shall include the software entities (CDDs, SWCs, standard AS modules, MCALs) which are directly dependent to the safety goals or feared events defined for the project. QM part shall accommodate the rest of the SW.

Assuming that the separation is already available, the goal is to ensure that our ASIL software will not malfunction because of a failure inside the QM part. ASIL software entities damaging ASIL part is not considered – it is supposed that ASIL software entities are developed according high quality level (100% coverage, critical code reviews and so on ...). The goal we have to fulfill is described in the picture below:



Interference from QM part is considered in 3 main aspects:

- **Timing and Execution** - with respect to timing constraints, the effects of faults such as those listed below can be considered for the software elements executed in each software partition:
  - Blocking or delay of execution
  - Components execution deadlocks
  - Incorrect allocation of execution time

Mechanisms such as cyclic execution scheduling, fixed priority based scheduling, time triggered scheduling, monitoring of processor execution time, program sequence monitoring and arrival rate monitoring can be considered.

- **Memory** - with respect to memory, the effects of faults such as those listed below can be considered for software elements executed in each software partition:
  - Data corruption
  - Unintended overwriting of memory allocated to another software component

Mechanisms such as memory protection, parity bits, error-correcting code (ECC), cyclic redundancy check (CRC), redundant storage, restricted access to memory, static analysis of memory accessing software and static allocation can be used.

- **Exchange of information** - with respect to the exchange of information, the causes for faults or effects of faults such as those listed below can be considered for each sender or each receiver:
  - Repetition of information
  - Loss of information
  - Delay/miss of information
  - Insertion of information
  - Incorrect sequence of information
  - Corruption of information

Our goals can be summarized in the following table, which describes what our system shall ensure:

First Level Safety Goal			Second Level Safety Goals		
SG ID	SG Definition	ASIL	SG ID	SG Definition	ASIL
SG_1	<i>The Turing software platform shall provide facility to ensure freedom from interference between partitions with different ASIL.</i>	ASIL B	SG_1.1	Safety relevant data must not be overwritten	ASIL B
			SG_1.2	Safety relevant runnables shall be executed within preliminary defined deadlines	ASIL B
			SG_1.3	Protected data elements are passed through to the receiver in correct order and with correct values. If a fault is detected the receiver gets notified	ASIL B

Possible safety mechanisms will be analyzed in the following sub chapters. However, in order the system to be ready for introduction of these mechanisms, there are some principles that have to be followed. They will facilitate the so called Freedom From Interference (FFI).

### 8.7.1 General design principles

It is advisable for the SWA to consider the following principles:

Error! Reference source not found., Part 6– Table 3 - Principles for software architectural design		Strategy for covering
1a	Hierarchical structure of software components	Software architect will assign the SWC to the respective layer – BSW, application or RTE
1b	Restricted size of software components	The size of the software component is restricted to the number of requirements it implements. The restriction is handled during the process of allocating safety



		requirements to a component by the architect.
1c	Restricted size of interfaces	The restriction is handled during the architecture design, when the architect is defining the interfaces. No other is allowed to introduce new interfaces.
1d	High cohesion within each software component	The high level of cohesion for components by design patterns (Model-View-Controller) for application level.
1e	Restricted coupling between software components	The coupling property is considered during architecture design and only software architect has the right to couple the components during design
1f	Appropriate scheduling properties	It is reasonable the complete scheduling is done with respect to the ASIL runnables in order to fulfill the FFI requirements for time and memory.
1g	Restricted use of interrupts	<p>The requirements will be satisfied in the way:</p> <ul style="list-style-type: none"> <li>- It is highly recommended to use interrupts category 2 and category 1 interrupts shall be avoided ;</li> <li>- when the category 1 cannot be avoided, the handler shall be developed according to ASIL B, and respective metrics for stack consumption and CPU load shall be reconsidered for all OS objects.</li> </ul>

### 8.7.2 End-to-End communication (E2E)

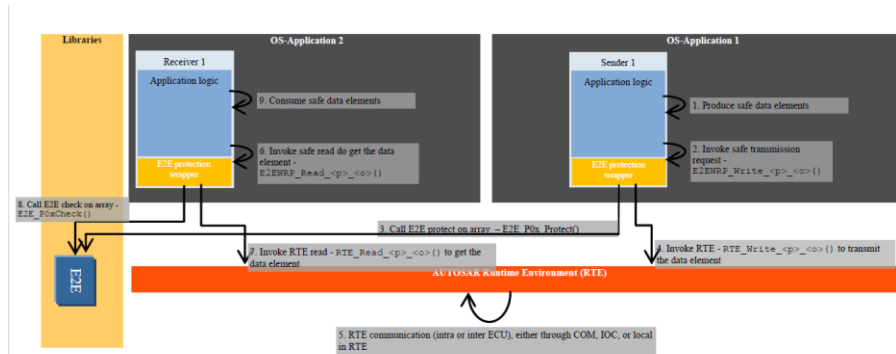
End-to-End communication satisfies our SG\_1.3. E2E is typically used to carry safety data over unsafe SW. This is the case with Autosar communication channels for example: Safety data from SWC on one ECU transmits safety data to SWC on another ECU – the data is safe, but the communication channels – not.

Several options for implementing E2E exist. We will review 2 options, which are defined by AS standard. The project can implement its own solution if the standard E2E approaches do not fit project's requirements or business case.

#### 8.7.2.1 End-to-End Protection Wrapper

Applications using the E2Elib or similar communication protection mechanisms have one major problem: the E2E library protection routines need the I-PDU representation to apply protection mechanisms.

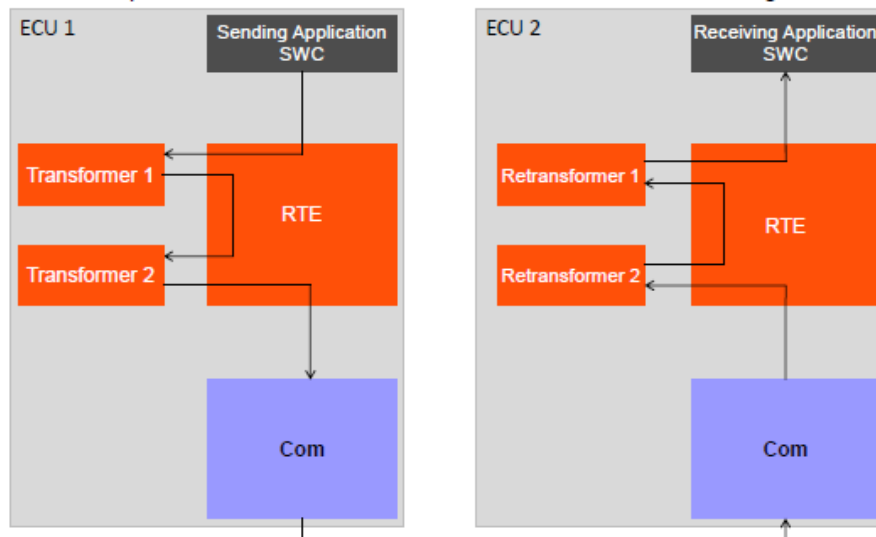
The E2E Protection Wrapper (E2EPW) provides a layer that circumvents this problem. The protection wrapper is generated code. That code consists of copy routines which know about the I-PDU layout and contain calls to the E2Elib routines. For transmission, the application passes the data element (DE) to the wrapper instead of the RTE. The wrapper then builds the I-PDU representation, invokes the E2E library function, and then the RTE function. For reception, the application calls the wrapper instead of the RTE. The wrapper receives the DE from the RTE and invokes the E2E library function before returning the DE.



### 8.7.2.2 End-to-End transformer

Key points:

- The E2E transformer ensures a correct communication of I-signals through QM communication stack. The communication stack is considered as “black channel” communication.
- The E2E Transformer is responsible for the invocation of the E2E Library based on the configuration of specific data element (I-signal).
- The E2E Transformer instantiates the E2E configuration and E2E state data structures, based on its configuration. All E2E profiles may be used to protect data.
- The E2E Transformer encapsulates the complexity of configuring and handling of the E2E and it offers a standard Transformer interface. Thanks to this, the caller of E2E Transformer does not need to know the E2E internals.
- The E2E Transformer is invoked by RTE, and the RTE invocation is a result of invocation of RTE API (read, write, send, receive) by software components.



Additional information for E2E can be found in the following sources:

- AUTOSAR\_SRS\_E2E.pdf
- AUTOSAR\_SWS\_E2ETransformer.pdf
- AUTOSAR\_SWS\_E2ELibrary.pdf
- UserManual\_E2EPW.pdf
- TechnicalReference\_E2E.pdf

### 8.7.2.3 Decision

Turing platform uses AUTOSAR delivery by Vector Informatik GmbH. At the moment of introducing E2E into the project, only E2EPW solution is available.

End-to-End transformer is new solution for AUTOSAR 4.2.1 (it is not present in previous releases). This is the reason why it is not present in Vector delivery. As a result of this benchmarks in terms of load and ROM/RAM consumption cannot be provided.

End-to-End transformer seems the better solution, since it abstract SWCs from the information that an E2E protected signal is transmitted. Thus SWC stays the same, no matter if the signal transmitted is E2E protected or not. Although this fact, Turing choice is to implement E2EPW, since this is the only ready-to-integrate solution. This decision does not force the customer projects to use the same approach. They can choose E2E Transformer or homemade solution.

### 8.7.3 Partitioning

RTE provides mechanisms for using OS Applications. ASIL related software components shall be allocated to the Trusted OS Application, the rest of the components to the Non Trusted. Different MPU setting can be set up for both. The nominal use case is to ensure that static data (RAM data), belonging to the Trusted OS Application, is not going to be overwritten by the rest of the SW. This is how we achieve our safety goal SG\_1.1, described in the beginning of [Safety](#) paragraph.

#### 8.7.3.1 Allocation of SWCs to Partitions

The first thing to be done when system with 2 OS Applications is used is to define what component goes in which Application. That's why analysis on the defined safety goals shall be performed. The output of this analysis shall be the list of components that are directly dependent for satisfying the defined safety goals. Once this list is ready, several points shall be taken into account for further analysis:

- Component shall belong to a single OS Application – this means that all the runnables realized by the component shall be mapped to tasks which are mapped to only one OS Application
- Trusted OS Application runs in CPU System Mode, Non Trusted in CPU User Mode.
- C/S communication between OS Application is expensive – Context switch is required in this case. This means CPU Load. Design shall be structured in such a way that these connections are as less as possible!

In order to minimize the C/S communication between the Applications the designer can adopt the following mechanisms:

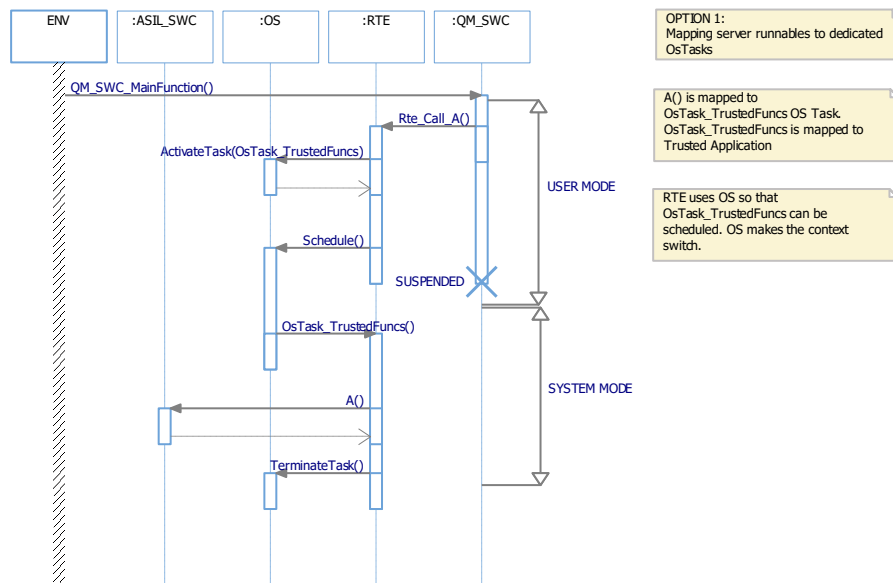
- Replace C/S communication with S/R - S/R communication does not require context switch.
- A component that is not directly dependent to the defined safety goals can be also assigned to the Trusted Partition – this can be used when several ASIL SWCs depend on QM SWC. By "depend" we mean C/S communication. Then this QM\_SWC can be assigned also to the trusted Application. Thus we "save" the context switch time and CPU load is lower. The price to pay is maintaining QM\_SWC as ASIL component – coverage, critical code reviews, limitations on source implementation i.e.
- A component can be duplicated – this may be used for variety of reasons:
  - Only part of the component functionality may be safety related. Example IoHwAb – provides services for reading Adcs, Digital Inputs, for writing Pwms. The safety functionality we need inside is to set a physical TT. That's why we may have IoHwAb and SafeIoHwAb.
  - Component calls server runnables from QM component. Example TtView – component that lit the TTs on the device. Some of the TTs require time base services, but our safety TT is not requiring this => creation of SafeTtView, which will not depend on time base services and will handle only the safety related TT.

#### **8.7.3.2 Inter Os Application communication**

S/R communication between Applications is OK. Problem appears when C/S is used, because it requires context switch. The following points describe possible ways to do this:

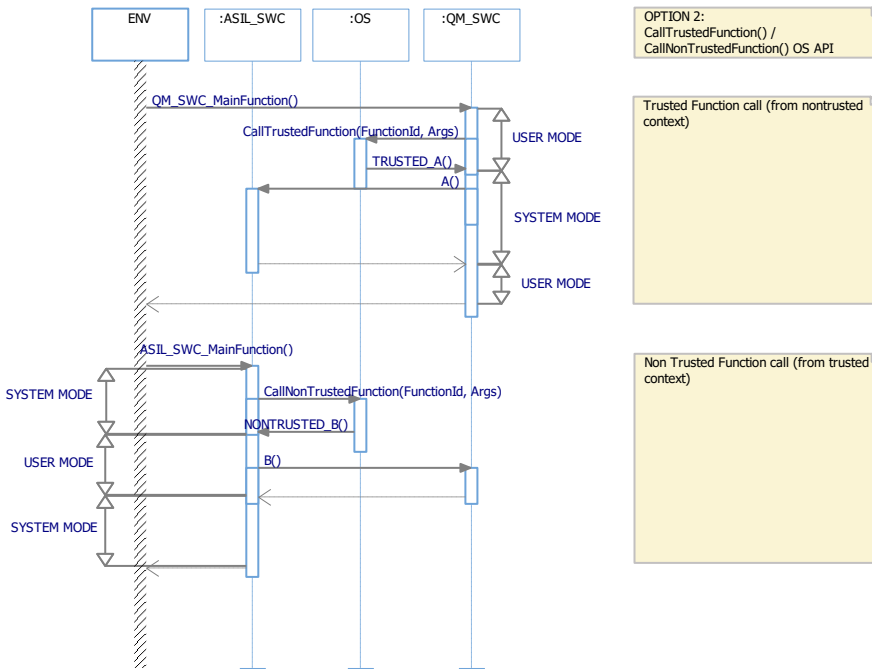
##### **8.7.3.2.1 Implicit**

With this approach, RTE and OS handle the context switch automatically. All that is needed as configuration, is to map the server runnable to dedicated task. In this way when the server runnable is called, RTE uses OS to schedule the OS Task accommodating the server runnable:



## 8.7.3.2.2 Explicit

Using this approach the client shall take care for the context switch. When server runnable, residing in different OS Application, is about to be called, client does not USE Rte\_Call API, but switches the context first through OS API CallTrustedFunction() or CallNonTrustedFunction(). OS handles the context switch and transfers the arguments of the call. As a result a runnable is called which runs in the changed context. From this runnable we can use the Rte\_Call API to call the real server runnable:



## 8.7.3.2.3 Home-made solution

It's not forbidden to use specific solution, fitting the specific project needs.

## 8.7.3.2.4 Decision

SWA recommends letting RTE and OS do the necessary activities automatically. This is the so called [Implicit](#) communication. The advantage is that client SWC implementation is not dependent to weather server runnable from the same or other partition is called. Additionally all protection mechanisms are handled by RTE and OS modules.

However this does not mean that SWA forbids the usage of the rest of the options. There are situations where other solution is needed. Example for this is an ASIL SWC calling many server runnables from QM SWCs. If the "Implicit" option is used this means twice context switch for every Rte\_Call. If option "Explicit" is used, calling SWC may switch context explicitly only once. In the function to be called by OS we have already changed the context. Now we can have calls to all server runnables without explicit context switch.

### 8.7.4 Graphics subsystem safety

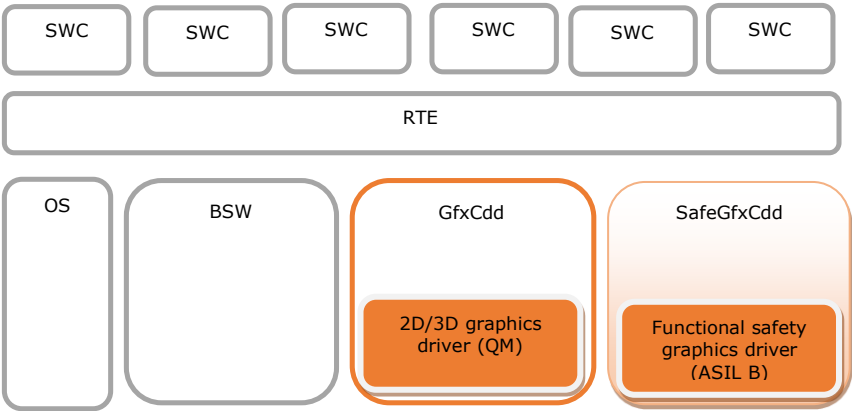
There are 2 general strategies for satisfying safety needs for the graphical subsystem. The first one consist of introducing a safety redundant path for displaying safety relevant elements. In short we can call this option "Safety stream". The second strategy is to have only a single display stream and to monitor if graphic element is correctly displayed.

#### 8.7.4.1 Safety stream

Copies of this document are uncontrolled

Page 86 of 119

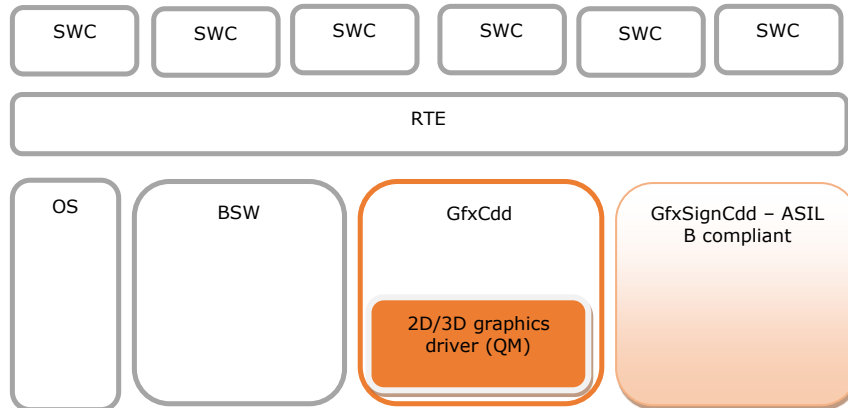
Deploying this strategy means that a dedicated safety stream is provided for safety relevant graphics elements. This stream exists in parallel with the normal display stream. The global architecture look like this:



The architecture above assumes 2 components – GfxCdd, which will realize the normal display stream and which will be used for all graphical elements except the ones that are safety relevant. SafeGfxCdd provides the same functionality, but uses specific driver developed to meet ASIL B requirements. This is the component to realize the safety stream. It is supposed to run in a trusted context and to have all its static data protected by MPU.

**8.7.4.2 Signature check**

This strategy does not provide safety path for displaying data. Instead it relies on the normal display channel to visualize graphical elements. Safety is achieved by monitoring for errors. To achieve this, special peripheral named “Signature unit” is used. When it is used, CRC is calculated on a specified displayed area and it is compared against a reference value. The assumed architecture looks like this:



GfxSignCdd is the Cdd to realize the management of the signature unit peripheral.

#### 8.7.4.3 Decision

“Safety stream” seems the better solution since it ensures safe display path. It provides the so called “active safety”. Problem with it is that the safety graphics driver is not available at Visteon site. It is not available on Cypress side either. This means that in order to use it Visteon has either to develop it OR to purchase it. In both cases solution will be available after a significant period of time.

After conversation with Turing safety engineer, it was clarified that monitoring for correct display of elements also satisfies ASIL B needs. Since this is acceptable and weigh easier to implements (also less risky) Turing choice is to use [Signature check](#) option.

#### 8.7.5 Peripheral protection unit (PPU)

PPU is a peripheral which provides access to peripherals in CPU user or/and privileged mode. The use of PPU becomes mandatory in case the system uses [Partitioning](#) mechanism. This is true because some of the peripherals are used by safety components (that run in privileged mode), while others are used by QM components (that run in user mode). The functionality can be implemented in a driver bloning to the MCAL layer – PpuDrv.

What is aimed with introduction of this module is to protect peripherals, exclusively used by a trusted components, from access by non-trusted components. For details about PpuDrv refer to the Turing UML model.

#### 8.7.6 Other safety mechanisms

##### 8.7.6.1 Watchdog manager (WdgM)

Watchdog manager shall be used in order to satisfy safety goal SG\_1.2, connected with timing and execution constraints. Since this is standard AS module, no additional design for it is assumed. The way it is going to be used is project specific. However typical usage is to monitor periodic OSTasks belonging to trusted OS Application:

##### 8.7.6.2 Error correction code (ECC)



ECC shall be used to guarantee safe code execution. Since ECC is ON by default on Traveo micros, no specific driver for this is assumed.

#### 8.7.6.3 Clock monitoring

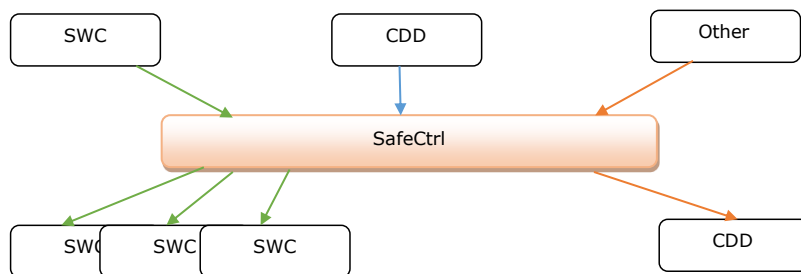
Clock monitoring could also be used to guarantee that certain timing is correct. Typical usage of it is to monitor clock feeding the OS tick runs appropriately. Since this is native configuration of the Mcu MCAL module, no additional design is assumed.

#### 8.7.7 Reaction on safety failure

2 strategies exist to handle failure due to some of our safety mechanisms:

##### 8.7.7.1 Centralized handling

With this approach failures are reported to a third component, let's say SafeCtrl. This component contains a lookup table with safety failure id and corresponding action. When it is informed about given failure, the component could impose specific behavior to other component or set of components.



##### 8.7.7.2 Distributed handling

According to this approach, centralized management of safety failures is absent – decisions are taken by the SW asset that is directly informed about the failure.

##### 8.7.7.3 Decision

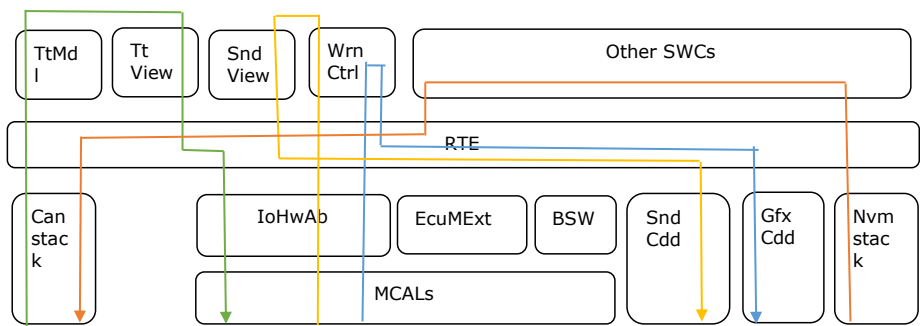
What has to be done on safety failure is project specific. SafeCtrl has too many specific parts – safety failures, actions. Actually only the presence of the component can be “bookshelf”-ed.

#### 8.7.8 Example design

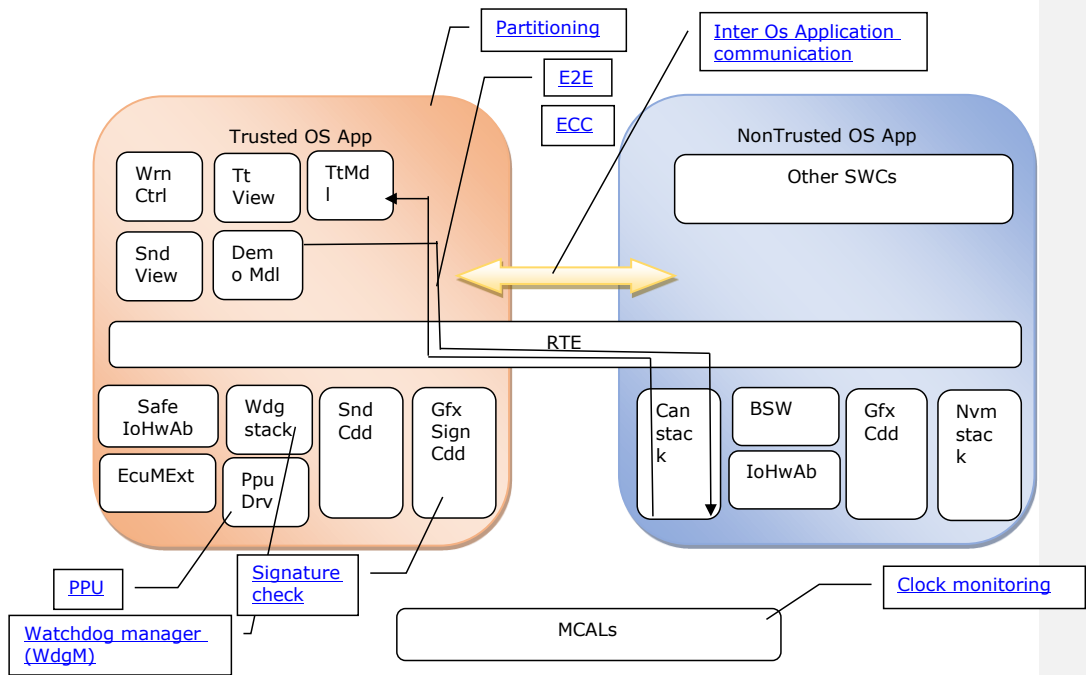
Let us assume we have the following safety goals:

- SG1: Physical telltale (trigger condition: CAN signal)
- SG2: Graphics element (trigger condition: HW input)
- SG3: Safety sound (trigger condition: the same HW input as SG2)
- SG4: Outgoing CAN signal (Data read from Nvm)

Let us assume the following architecture and the impact of the defined safety goals:



Taking into account the safety mechanisms we have discussed, proposal for design like the one below can be made:



8.8 I/O hardware abstraction

8.8.1 IoHwAb

Rules and constraints for detailed design of IoHwAb.

R.IOHWAB.001

This component is dedicated to manage at low level all inputs and outputs → digital, analog, and possibly PWM logic on top of Autosar PWMDRV.  
The goal is to avoid creation of other CDDs, dedicated to management of I/O.  
Good practices and lessons learnt to be taken from existing projects.

#### R.IOHWAB.002

The placement of the component in the architecture for TA and for TB is the same → up to RTE and down to MCALs as they are defined by Autosar.

#### R.IOHWAB.003

As far as HW is the same, and configuration is the same, TA and TB must be able to share the same component.

#### R.IOHWAB.004

Component must be split in the next parts, separately usable:

- DI – Digital inputs, configurable, reusable
- DO – Digital outputs, configurable, reusable
- AI – Analog inputs, configurable, reusable
- AO – Analog outputs, configurable, reusable
- PWM – Pulse-width-modulation management, reusable
- Manual user-defined implementation → blocks inside are structured to ease the works with the reusable parts above.

Note: More parts might be added during the detailed design. To be checked for **Input Capture** and Timer Outputs.

#### R.IOHWAB.005

Parts like DI, DO, AI, AO should be structured to fulfill these:

- Have management of several inputs/outputs.
- All inputs/outputs should be able to have the same functionality → configurable. I.e. equal treatment is required.
- Filters and/or de-bounce mechanisms should be able to be attached to inputs/outputs.
- If filters are configured, manual part must be able to pre-hook and to post-hook the filtering with user defined code.

#### R.IOHWAB.006

Design should propose how to manage second level logic that needs to deal with mixed I/O. For example 2 DI + 1 DO + 1 AI.

#### R.IOHWAB.007

Interface to RTE, where appropriate, must return status of particular input channel:

- VALID
- INVALID, means returned value must not be used
- NO\_VALUE, means returned value must not be used

Mapped properly to Std\_ReturnType constants.

#### R.IOHWAB.008

Special attention to be paid to HW peripheral malfunction and I/O signal management. Error handling per type of HW to be described.

#### R.IOHWAB.CFG.001

The following levels of configuration must be ensured:

- Multi-product variants, run-time selectable during the component initialization only. The use-case is: During system initialization the concrete product variant is selected, then IoHwAb is initialized (via SetCfg()) or reading something from RTE or hard-coded ). This could possibly affect several parts of the component. Detailed design must offer strategy/guidelines to handle this.

- Channel configuration, run-time selectable even after initialization is done. Diagnostics might use this.
- Filter configuration, design-time defined and selected once at run-time during the initialization. For example, it won't be possible to change  $\tau$  of PT1 filter when channel is running.

#### R.IOHWAB.CFG.002

The next things, once configured, must not be able to be changed:

- Number of product variants
- Number and content of the predefined settings for each product variant
- Sequence of filters, assigned to a particular channel
- Individual filters in the sequence cannot be enabled/disabled at run-time. But the whole channel sequence could be changed with another one.

#### R.IOHWAB.CFG.003

Initially configuration can be done manually. Long term, there must be configuration tool, following the rules for such tools, described above.

#### R.IOHWAB.FILTERS.001

There are two main categories of filters that needs to be supported → digital and analogue.

The following frequently used ones needs to be maintained in the bookshelf:

- High-pass
- Low-pass
- Hysteresis
- Inverter
- Average window
- Average moving window

De-bouncing to be added.

#### R.IOHWAB.FILTERS.002

Also, the user should be able to define own filters and to attach them to the I/O channels.

#### R.IOHWAB.FILTERS.003

Up to 3 filters must be able to be attached to every single channel to be executed consecutively within one call.

There must be a way to implement monitoring/reporting of the intermediate values between the filters.

#### R.IOHWAB.SAFETY.001

Reusable parts that will be designed must be able to be used by two instances of the IoHwAb → One non-safe and one which is safe (named *IoHwAbSafe*). The idea is that safety part:

- Can be allocated in a separated memory partition.
- Its main periodic task will be monitored by safety OS features → Execution, stack usage.
- Additional protections will be implemented only for safety channels, so the overhead (RAM/ROM/ CPU) will not be put over the non-safety parts.

#### R.IOHWAB.SAFETY.002

Specific functionality must be ensured for measurement of I/O channels. Usually for safety telltales we need to measure bulb failure, which is a *fear event*.

#### R.IOHWAB.SAFETY.003

Periodic refresh of outputs might be needed. To be described how to do.

Special attention for the next release:

- Investigation to be made to have this as *ManagedComponent*. Life-cycle here is needed.

Body Controller requirements to be evaluated and added to these above.

## 8.9 Timing services

### 8.9.1 TmExt

This chapter explains the TA and TB needs regarding time services that has to be covered. They are divided into 3 categories:

- SW timers used on polling during execution of a periodic runnable.
- SW timers which notify the user for expiration via callbacks.
- SW pulses generation → programmatic + callback notifications of front and back edges.

On top of those it is planned to obey all the requirements that were collected and approved by A2TOM::TIM framework.

#### R.TMEXT.001

A dedicated service extension is defined. Valid for TA and TB versions.

The name of the component is intentionally selected. The first part is the standard name of the Autosar 4.2.1 service *Tm*, which is doing similar things. The second part *Ext* says that this is extension of it. This approach is already used for several other Autosar extensions, so we keep the pattern.

Because this is actually BSW service, it can be used in other TB BSW components too.

#### R.TMEXT.002

One of the most important roles of that component it to provide the *same time base* to all SW time services.

The main idea is that all time related services should be derived from the same time base, to share the same common code and to be configured in one place → i.e. all needed time services for a particular user has to be described once in one place.

#### R.TMEXT.003

SWCs and CDDs in derived projects are able to measure time intervals in a uniform way. This is to ensure portability between various systems. The service extension component, named *TmExt*, is dedicated to provide 16-bit and 32-bit timers in 1 [ms] resolution. Investigation was made to prove that this resolutions is ok:

- TA 4.2.1 offers 100 [μs] and 1 [μs] resolutions → these might be ok for Body controllers, but not really needed for Driver Information and Infotainment functionality:
  - If there is a use-case of a cluster that is mix between Body and DI features, it still will be able to use *TmExt* by changing the configuration to 100 [μs]. This will decrease the maximal timer values that can be used.
- All SOSA and xJCI Gen2 projects are using max 1 [ms] timers. No requirements were found that tend to use higher resolution.
- Discussions with graphical experts were done. They also measure time in 1 [ms]. The FPS required nowadays is 60, BMW requires 75 and we supposed that in 2..3 years somebody might want 100. Even in that case 1 [ms] is enough.
- CPU load metrics can be done with GPT with higher resolution, used only in measurement build. For example with the native Tm service as proposed by Autosar or hand coded.

SWCs and CDDs have the need to use SW timers. Frequently more than one at the same time. The range of these timers is up to 1 day in most of the cases. Longer time intervals are measured with different services from different components that support time/date/calendar – usually called Model for clock/calendar.

#### R.TMEXT.004

Copies of this document are uncontrolled

Page 93 of 119

*TmExt* must ensure exactly one service for active waiting for short intervals in [ns] with 16 bit parameter → up to 65.535 [ms].

This is because the standard Autosar 4.2 Tm service is not available (won't be purchased) and we have TB version that also needs this.

For sure the implementation of such services must take into account compiler optimization, pipeline and similar to be precise. Prove with oscilloscope to be done.

#### R.TMEXT.005

Notification via callbacks to be investigated. RTE features to be used to distribute those notifications.

#### R.TMEXT.006

The requirements and functionality of xJCI SRV\_TimeGen to be analyzed and improved, if needed, and embedded into *TmExt*.

#### R.TMEXT.007

Separated functional blocks should be able to be excluded when not needed.

### 8.10 GUI sub-systems

Under construction. Working together with HMI team.

### 8.11 System mode management

#### 8.11.1 The problem

The projects across the company are using the term *Mode Management* to denote various things, not directly related to each other. Frequently seen problem is to have a mixture between the definitions, coming from OEMs and internally defined things. Ex.: Power mode management. There is a cultural difference between xJCI and Visteon:

- Each company has own definitions/understandings
- These do not match entirely

**A clear definition is required.**

#### 8.11.2 Analysis of the needs

The mentioned problem obviously has several aspects.

To reveal the needs the next questions needs to be answered:

- How many different things are called '*mode*'?
- For which of them platform must provide solution or at least guidelines?
- What are dependencies between different *modes*?
- What does it mean to *manage* the *mode*?

List of frequently used '*modes*' in various projects:

- Key position modes ( Ex: OFF, ACC, IGN, CRANK )
- Car modes (Ex.FACTORY,TRANSPORT,NORMAL,CRASH)
- Voltage modes (Ex.NORMAL, UNDER-,OVER-VOLTAGE...)
- Degraded modes (Ex. NORMAL, LIMP-HOME...)
- Safety mode (Ex. Safe state after feared event happened)
- Secured modes (Ex. Diag. sessions, SHE related)
- Life-cycle control is called sometimes mode management.
- ...more might come into the game.

---

Copies of this document are uncontrolled

Page 94 of 119

The behavior of the device under these *modes* depends on:

- OEM specifications
- Architectures/ Platforms constraints (like Autosar)
- HW to SW Interface document
- High-level design decisions
- 3-rd party components
- Detailed design and implementation of home-made components

Projects are implementing these in almost chaotic way - whenever something pops up. Consistent and structured approach needed to overcome collisions and to reduce the time to get a mature product!

In all the cases, all *mode kinds* are perceived/implemented as enumerators, consisting of *modes*, denoting individual modes of that kind.

Transitions between mode literals of a particular kind are also defined - what is possible under which conditions.

In the specifications and in the design *modes* of particular kind are represented usually as state-machines.

The needs of the projects are to identify clearly all involved state-machines for modes and how they are related to each other.

Once the set of *modes* is decided for a particular project, every SW component in the system must know when what to do:

Component functionality	Mode1:E1	Mode1:E2	Mode2:X1	Mode2:X2	Mode2:X3
F1	X		X	X	X
F2		X		X	
F3	X		X		X

High-level and detailed design are significantly affected by this!

To make this differently for each component is nonsense. For example reuse will be lost, risks of regressions, etc.

The need is to set up the right uniform approach since the beginning.

### 8.11.3 Requirements & constraints

- OEM definitions for *modes* should be preserved and maintained over the time. Specs vary.
- The home-made defined *modes*, depending on SWA decisions, must not collide with OEM ones.
  - They must be separately defined/ managed
  - They must be extendable
- Because the number of *mode kinds* is usually > 3, and everyone has number of *modes*, the variants of component behavior is significant. The complexity must be controlled, i.e. special care to be taken during the design.
- Because of the hybrid nature of the projects (home-made+ 3<sup>rd</sup> party + OEM constraints) unique solution is not possible.

#### 8.11.4 Definition of mode management

*Mode* is a particular state of a state-machine, dedicated to one kind of functional control at system level.

System life-cycle and component life-cycle are NOT branching of functional control, so they are NOT considered as modes.

*Mode management* is about all the following points:

- Run-time elaboration of all needed *kinds of modes*, including:
  - Solving the possible collisions between the sources indicating the same mode.
  - Solving of the possible collisions between different modes.
- Distribution of the elaborated modes to SW components
- Rules and recommendation how individual SW components should use the distributed modes.

#### 8.11.5 Alternatives

(A1) No rules. To leave individual components to produce and consume modes as they wish:

- Simply not acceptable. Unstable & uncontrolled.

(A2) To separate mode producers and consumers and to define in which way the produced mode is accessed by consumers.

- + Problem is divided in well-defined domains (production, transportation, consumption), everyone can be tackled individually and can be done in several ways.
- + Clear and maintainable
- + Extendable
- + Distributable
- 3<sup>rd</sup> party code can create obstacles

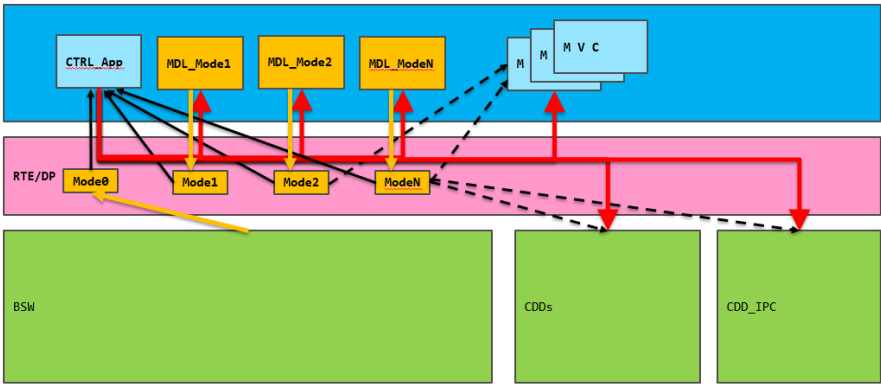
The chosen one is (A2).

#### 8.11.6 Decisions

- Use MVC to achieve the separation of producers and consumers. Models must produce the mode.
- Use DP in RTE to:
  - Store the produced modes
  - Share modes across cores and memory partitions
  - Protect current modes from preemption
  - Ensure notifications on change and/or consumption, if needed
- Use IPC to replicate the produced modes to other MCU (GIP for ex.)
- Ensure exactly one producer for each mode at a time, even with multi CPU
- Ensure two levels of management:
  - System level → with the help of component life-cycle
  - Component level → dealing with modes inside components

#### 8.11.7 MVC Solution





Legend: → Component life-cycle management + Degraded level notification  
→ Mode production  
→ Mode reading

8.12 Synchronized activation

There are several cases which need synchronized activation of features. The main directions are described here.

Warnings synchronization

See the dedicated chapter below.

Blinking synchronization

See in UML the detailed design of *TmExt* component, *Pulse* functional block. It is especially designed for this purpose.

For sync with graphical telltales (called pictograms) GUI system receives notification from TmExt. Normally till next VSYNC (usually 16 [ms]) the display lists are processed and pictograms start blinking. We have lower threshold of 30 FPS → 1000/30 = 33.(3) [ms]. The closest power of VSYNC period is 2 \* 16 = 32 [ms]. This is the max de-sync between telltales and display pictograms. For Turing, we are checking whether it is acceptable from user perspective or not. If not, we must improve.

Commented [IDP1]: 16 TBC

Backlit synchronization

There is bad visual effect to the driver, caused by backlit PWM frequency, which is not power of VSYNC frequency of the TFT. Lessons learned from Gen2 platform, used in several project. The solution is to apply this:

$$F_{pwm} = F_{vsync} * K$$
 , where  $K \in [3, 4_{default}, 5]$  → parameter in NVM needed in some cases.

This must be taken into account during the design.

HMI synchronization

HMI sub-system of the device includes all visible and audible to the driver features:

- Backlight of display, gauges, scales, telltales, etc...
- Sounds
- Telltales
- Graphic

All of them have separate HW channels and dedicated views to manage them. There are particular states of the ECU which insist presentation channels to act synchronously. Turing SWA manages this introducing dedicated SWC – **HmiCtrl**.

#### R.HMICTRL.001

**HmiCtrl** is responsible for detection of particular HMI states and their prioritization.

#### R.HMICTRL.002

**HmiCtrl** drives presentation channels via uniform interface(s). HLD must ensure possibility for feedback from the views.

#### R.HMICTRL.003

All synchronization requests are sent to all connected channels. It is up to views to react or disregard the request.

#### R.HMICTRL.004

Synchronization HLD is part of bookshelf functionality, but detection and prioritization of the states are project dependent features. Identified global features to be supported:

- Welcome/farewell sequences
- Engineering Test Mode

### 8.13 Debug and trace

#### 8.13.1 DLT

##### 8.13.1.1 AS Research

*DLt* is a generic Logging and Tracing functionality for any component connected to the RTE, Det and Dem. More detailed description could be found inside [R6]. Three types of *DLt* features are introduced inside:

- A. Communication with SWC
- B. VFB-Trace
- C. Dem and Det log

According to the description inside [R6] each SWC which uses *DLt* (variant A) should have at least 2 ports to be able to do its job:

- Dlt\_SetLogLevel – input for the SWC where it receives the level.
- Dlt\_SendLogMessage – C/S call, used by the SWC to send the data to the Dlt.

Picture below, taken from [R6] is showing this:

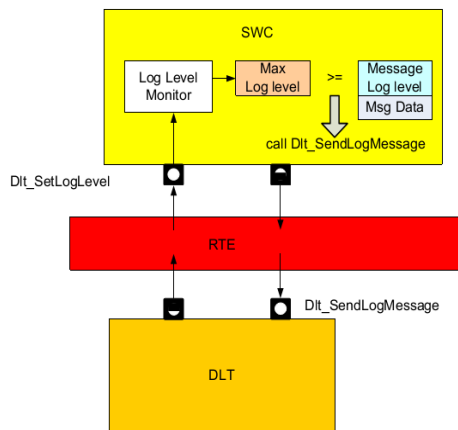
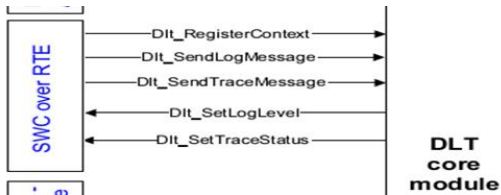


Figure 12 SW-C shall be aware of the status of log level and trace status

The picture with all the APIs for the SWC, also taken from [R6] looks as follow:



8.13.1.2 Delivery from Vector

The Vector delivery according to the [R7] contains the following:

Supported AUTOSAR Standard Conform Features
Log messages from DET
Log messages from DEM
Verbose logging mode
Non-verbose logging mode

Table 3-1 Supported AUTOSAR standard conform features

**Not Supported AUTOSAR Standard Conform Features**

Logging of errors, warnings and info messages from AUTOSAR SWCs, providing a standardized AUTOSAR interface
Gather all log and trace messages from all AUTOSAR SW-Cs in a centralized AUTOSAR service component (Dlt) in BSW
Trace RTE/VFB
Enable/disable individual log and trace messages
Control trace levels individually by back channel

The following features are provided beyond the AUTOSAR standard:

**Features Provided Beyond The AUTOSAR Standard**

DLT communication is based on XCP instead of the specific DLT protocol
Messages for non-verbose logging are stored in an A2L fragment file instead of FIBEX.

Table 3-3 Features provided beyond the AUTOSAR standard

The function `Dlt_SendLogMessage` is the implementation of the verbose log mode. Inside there is a call to the `Xcp_Event(DLT_XCP_VERBOSE_MSG_EVENT)`. This function could be used by the components to log their info.

**8.13.1.3 Project 35up IC**

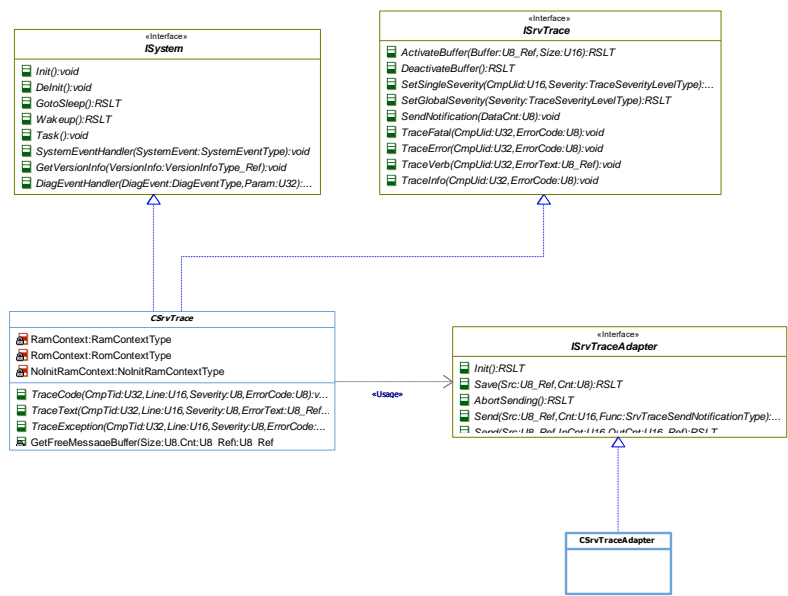
Inside 35up IC there is a BMW implementation of the Dlt. There are variants 'A' and 'C' of the described AS Dlt features.

**8.13.1.4 SRV\_Trace**

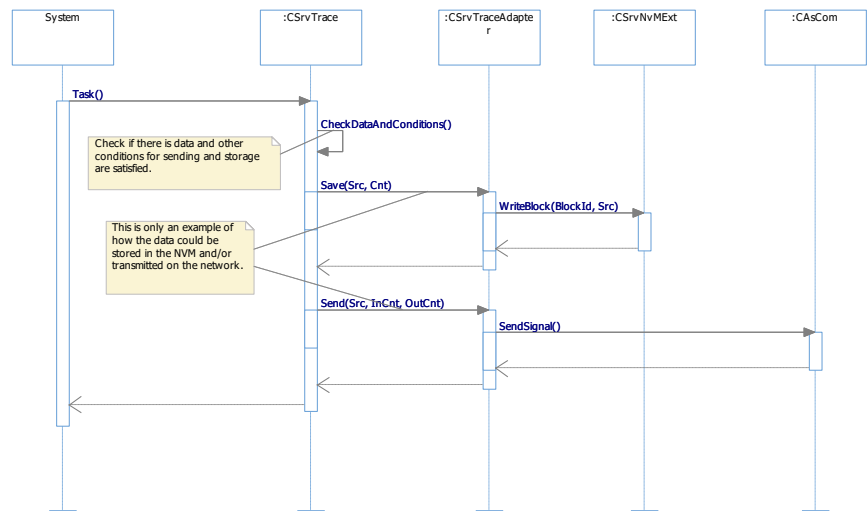
This is xJCI Gen2 component used for tracing. See [R8].  
Its public interface has the following service APIs:

Public interface function	Function inside the SRV_Trace implementing the functionality
<code>SRV_Trace_TraceError</code>	<code>SRV_Trace_TraceCode</code>
<code>SRV_Trace_TraceFatal</code>	<code>SRV_Trace_TraceException</code>
<code>SRV_Trace_TraceInfo</code>	<code>SRV_Trace_TraceCode</code>
<code>SRV_Trace_TraceVerb</code>	<code>SRV_Trace_TraceText</code>
<code>SRV_Trace_TraceWarn</code>	<code>SRV_Trace_TraceCode</code>

Class diagram of this component:



One possible usage:



Below could be found some examples of SRV\_Trace usage with explanations what the functions do:

→ **TraceFatal**  
Ex.: `TraceFatal(HWAL_SOUND_MODULE_ID, HWAL_SOUND_E_UNKNOWN_ERROR);`

```
// Used to store and trace messages of type code with the severity 'fatal error' in the BURAM. In
// addition it performs a reset.
```

```
void SRV_Trace_TraceException(uint32 u32CmpTidP, uint16 u16LineP, uint8 u8SeverityP, uint8
u8ErrorCodeP);
```

```
→ TraceCode
```

```
→ TraceInfo
```

```
→ TraceWarn
```

```
Ex.:TraceError( cXDR_IOC_MODULE_ID, cXDR_IOC_E_UNINIT );
```

```
//Used to trace messages of type code with the severities 'error', 'information' and 'warning'.
```

```
void SRV_Trace_TraceCode(uint32 u32CmpTidP, uint16 u16LineP, uint8 u8SeverityP, uint8 u8ErrorCodeP);
```

```
// Used to trace messages of type text (string).
```

```
void SRV_Trace_TraceText(uint32 u32CmpTidP, uint16 u16LineP, uint8 u8SeverityP, uint8 *pu8ErrorTextP);
```

Current MQB implementation does not use the severity feature of the SRV\_Trace, described inside [R8].

Structure of the SRV\_Trace buffer could be seen in the table below:

Size	Counter	Time	CmpUID	Line	Severity	Data
------	---------	------	--------	------	----------	------

Possible connections to Dlt:

A. Direct connect SRV\_Trace to Dlt:

There is possibility to connect SRV\_Trace to Dlt delivered by Vector. In this case all the trace messages will be sent via CAN over the XCP protocol. From Dlt perspective, SRV\_Trace will have only one ID. All the components connected to the SRV\_Trace will register their calls with the CmpUIDs and Line numbers. This info will be part of body of the final message.

Advantages:

- Current SRV\_Trace used in MQB is not connected to any communication mechanism => connecting it to the Dlt we will have a common standardized method for log and trace.
- xJCI SRV\_Trace clients(SWCs, CDDs) do not change their code.
- SRV\_Trace could be used for TB without Dlt.

Disadvantages:

- There will be double buffering. Once in Dlt and once in SRV\_Trace.
- In the TB we do not have Dlt and XCP => XCP replacer with reduced functionality (only sending) could be created. Direct connection to CAN should be created, if log/trace message should be sent outside.
- The messages are sent from SRV\_Trace main function.

B. SRV\_Trace to be connected directly to XCP(possible only for TA):

This way the double buffering could be avoided.

C. SRV\_Trace connected to DCM:

In this case read by diagnostic message could be used in order to get the info stored inside SRV\_Trace buffer.

#### 8.13.1.5 Lib\_Assert

Lib\_Assert is other possibility used mainly inside Gen2/Gfx. The *assert routine* could be infinite loop in DEBUG compilation, which is causing a reset if Wdg is not disabled.

#### 8.13.1.6 LogFlow class from MQB project

This is C++ implementation of the classes LogFlow which uses the class RollingBuffer for the implementation of logging mechanism. It was used in MQB. They are storing the input info in 4 buffers. Those buffers are dumped with the help of the debugger. Below could be found the declaration of both the classes:

```
class RollingBuffer
{
private:
    char *    _buffer;
    int       _head;
    const int _size;
protected:
    void setHead(int head);
public:
    RollingBuffer(char * buffer, int size);
    virtual ~RollingBuffer();

    void add(char * ptr, int count);
    void clear();
    int  getSize() const;
    int  getHead() const;
    char* getBuffer() const;
};

class LogFlow
{
public:
    enum LogChannel
    {
        eLOG_CHANNEL_0,
        eLOG_CHANNEL_1,
        eLOG_CHANNEL_2,
        eLOG_CHANNEL_3,
        eLOG_CHANNEL_COUNT
    };
    static const unsigned long BUFFER_SIZE;
    static const unsigned long MAX_LINE_SIZE;
private:
    static bool _channelEnabledStatus[eLOG_CHANNEL_COUNT];
    static RollingBuffer _buffers[eLOG_CHANNEL_COUNT];
    static unsigned int _messagesCount[eLOG_CHANNEL_COUNT];
    LogFlow();
    static void incMessageCount(LogChannel);
public:
    virtual ~LogFlow();
    static bool enableChannel(LogChannel);
    static bool disableChannel(LogChannel);
    static bool isChannelEnabled(LogChannel);
    static unsigned int getMessagesCount(LogChannel);
    static void log0( const bool addPrefix, const bool addNLSuffix, const LogChannel ChannelIdP, char
const * const PrettyFuncNameP, char const * const FormatP, ... );
    static void log1( const bool addPrefix, const bool addNLSuffix, const LogChannel ChannelIdP, char
const * const PrefixP, char const * const FormatP, ... );
    static void log2( const bool addPrefix, const bool addNLSuffix, const LogChannel ChannelIdP, char
const * const ClassNameP, char const * const FuncNameP, char const * const FormatP, ... );
};
```

#### 8.13.1.7 Common TA/TB proposal

SRV\_Trace could be used for log and trace of all components. The info out of there could be obtained in the following manner:

- Dump the buffers with the help of the debugger.
- The buffers could be read through the diagnostic request.

In parallel Lib\_Assert could be used in debug mode with infinite loop inside.

For TA, SRV\_Trace could be used with a direct connection to XCP. Dlt could be used for DEM and DET tracer.

For TB, if there is big necessity of sending the info over CAN, a simple event sender in the XCP manner could be created.

#### 8.13.1.8 Decisions

Considering the facts we have today:

- There is already usage of Dlt inside BSW delivered by Vector. We will have more for the future. It could be disabled in the generators, but cannot be kept disabled long term.
- Currently Dlt does not support important features that are needed in general. They will come one day.
- Dlt is complex, requires ports to be configured (increase the problem with RTE configuration and footprint, which is critical for TB) and logic to be implemented in every SWC/CDD. There will be overhead.
- Specific component and session identification needs to be supported → here we have collisions with other, previously used, ways of component identification.
- It will be very difficult to have all Dlt functionality in TB.

The next decisions are taken:

##### R.DLTEXT.001

A new component *DLText* will be created. It will be AS compatible and will handle SRV\_Trace functionality. Interface will be reviewed and improved/simplified to best match the needs of the SWCs and CDDs, w/o big overhead.

##### R.DLTEXT.002

DLText will be called by SWCs and CDDs directly, not via RTE. Later on when dual-core or memory partitioning take place, upgrade of that component must be done. Also we need to wait DLT implementation from Vector and to evaluate its overhead.

##### R.DLTEXT.003

Connection to XCP to be used for sending messages on event. This means that the implemented relations must be: DltExt → Dlt → XCP. So, DltExt will not do buffering, but will call the existing Dlt functionality.

Note: At a later stage and on demand (projects request), the platform could implement UART connection as alternative to XCP. Nevertheless, XCP remains the main recommended approach.

##### R.DLTEXT.004

Severity levels of SRV\_Trace needs to be revised and to be mapped to the existing ones of Dlt. Connection to DCM is to be established as well.

##### R.DLTEXT.005

DltExt component must provide assert functionality to be used by all components, developed in house. The next macros are mandatory to be supported with the exact names given here:

STATIC\_ASSERT(cond) → stops at compile time when condition evaluates to zero

FATAL\_ASSERT(cond) → means exceptional situation; device reset in debug and in release

DEBUG\_ASSERT(cond) → used to stop at breakpoint in debug only, does not exist in release

Copies of this document are uncontrolled

Page 104 of 119



### 8.13.2 DET

**Default Error Tracer** (DET) is AUTOSAR defined BSW component. Its purpose, interfaces and functionality are described in "**AUTOSAR Specification of Default Error Tracer**" document. As it is explained inside, there is no explicit specification of its implementation. Next integration guideline is based on VECTOR delivered implementation and documentation – "**MICROSAR DET Technical Reference**".

#### R.DET.001

Usage of **Extended Debug Features** (*Filters, Logging, Break handler*) of **DET** component is up to the project needs. Turing SWA does not provide any constraints or restrictions on this.

#### R.DET.002

If there is a need to transmit traced errors outside the ECU, then the preferable approach is to use standardized AUTOSAR connection to the DLT - **Dlt\_DetForwardErrorTrace**.

#### R.DET.003

It is up to the project to use or not to use the DET extension callout **Appl\_DetEntryCallout**. In this case some constraints apply:

- Whole related implementation must be in dedicated **Det\_Callout.c** file, which becomes part of DET component
- **Det\_Callout.c** must include **Det.h**
- Content of the file must be conditionally compiled only if **DET\_ENTRY\_CALLOUT\_ENABLED** is defined

### 8.13.3 RTM

**MICROSAR Runtime Measurement** (RTM) is VECTOR defined BSW component. Its purpose, interfaces, functionality, configuration and how to integrate are described in:

- **MICROSAR Runtime Measurement, Technical Reference, Version 2.00.00**
- **User Manual, AMD – MICROSAR 4, Version 1.2.1**

Turing SWA provides brief guideline how to configure the module itself and other parts of the system for the need of CPU load measurement.

#### Scope

RTM is used for measurement of the time spend into runnables, tasks and interrupts. It is also used for measurement of overall CPU load.

#### General configuration

Recommended settings for general configuration parameters of the RTM. Not marked settings are up to system integrator and project needs.

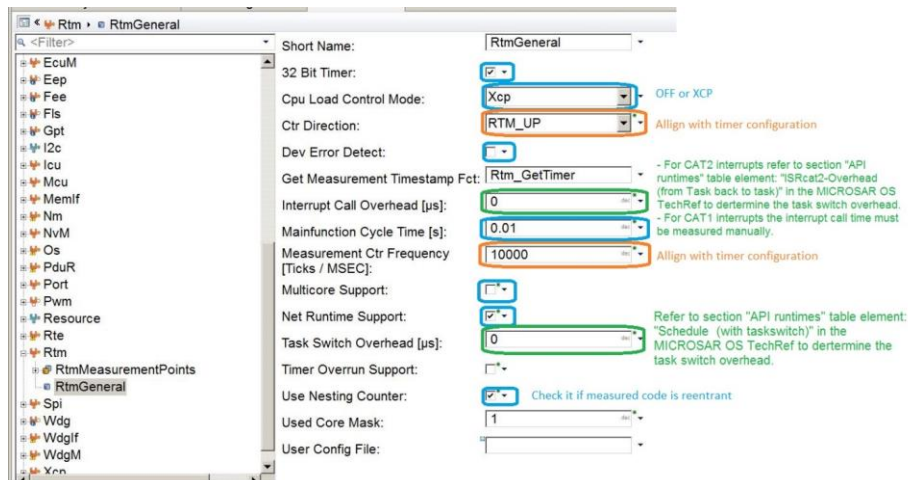
- **Blue** rectangles are recommended settings with their variants if any.
- **Green** rectangles to be configured according delivery information
- **Orange** rectangles must respect configuration of the used timer. Proposal is to use GPT timer configured in "*Continuous mode*" with resolution 0.1[us].

Remainder:

- If "**Cpu Load Control Mode**" is switched on, then configure related measurement point.

Copies of this document are uncontrolled

Page 105 of 119

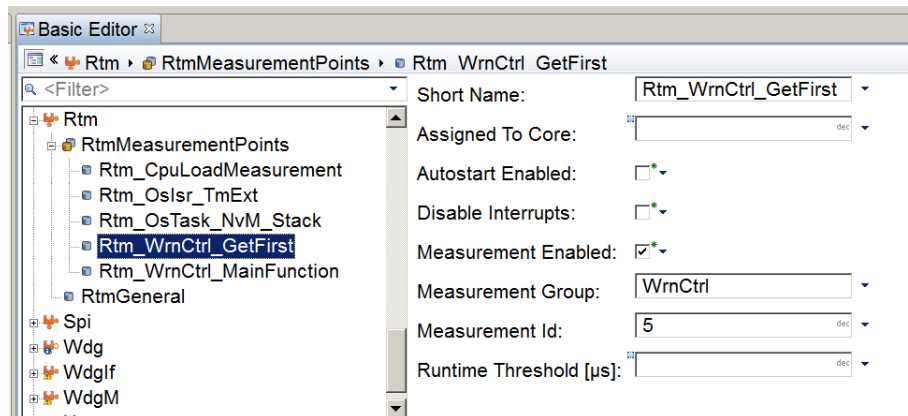


### Measurement point configuration

RTM needs one such measurement point configured for each runnable, OS task and interrupt. Special measurement point is needed if **Cpu Load Control Mode** is turned on.

Hint:

Put component name as "Measurement group" for all point related to this component.



### CAT1 Interrupts

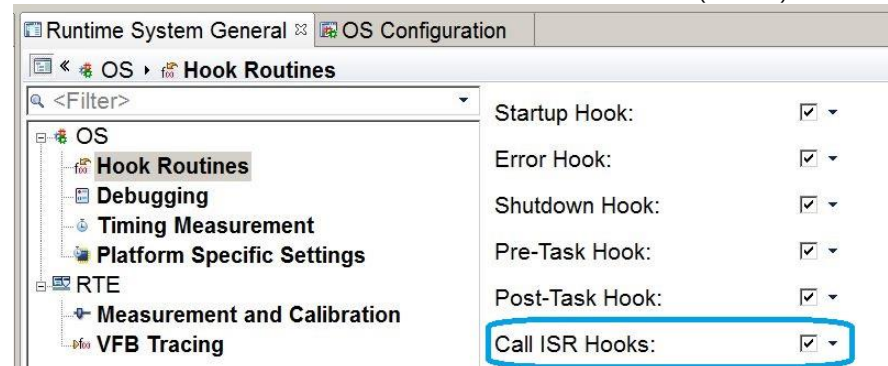
Measurement of the time spend inside CAT1 interrupts is based on manual instrumentation of handlers implementation. Recommended sequence is:

- 1) Call to **void Rtm\_EnterIsrFct(void)**.

- 2) Call to "**Rtm\_Start(RtmConf\_RtmMeasurementPoint\_<Measurement Name>)**", where **<Measurement Name>** is the name of the RTM measurement point configured for this interrupt.
- 3) Handler implementation
- 4) Call to "**Rtm\_Stop(RtmConf\_RtmMeasurementPoint\_<Measurement Name>)**", where **<Measurement Name>** is the name of the RTM measurement point configured for this interrupt.
- 5) Call to "**void Rtm\_LeaveIsrFct(void)**".

### CAT2 Interrupts

Measurement of the time spend inside CAT2 interrupts handlers is based on ISR user hooks configured inside "**DaVinci configurator**" tool. To enable them go to "*Runtime System General*" -> "*OS*" -> "*Hook routines*" -> "*Call ISR Hooks*" and set checkbox to **TRUE** (checked).



This will enable calls to:

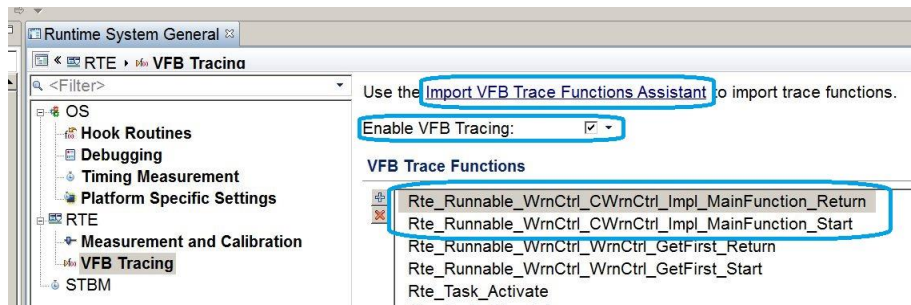
- "**void UserPreISRHook(ISRType x)**". Implementation of the hook must contain:
  - Call to "**void Rtm\_EnterIsrFct(void)**".
  - Call to "**Rtm\_Start(RtmConf\_RtmMeasurementPoint\_<Measurement Name>)**", where **<Measurement Name>** is the name of the RTM measurement point configured for interrupt identified by parameter "**ISRType x**".
- "**void UserPostISRHook(ISRType x)**". Implementation of the hook must contain:
  - Call to "**Rtm\_Stop(RtmConf\_RtmMeasurementPoint\_<Measurement Name>)**", where **<Measurement Name>** is the name of the RTM measurement point configured for interrupt identified by parameter "**ISRType x**".
  - Call to "**void Rtm\_LeaveIsrFct(void)**".

### Runnables

Time measurement of the runnables uses "**VFB Tracing**" functionality described into "**Specification of RTE AUTOSAR Release 4.2.1**". Trace events related to runnables are described in chapters "**5.11.5.4.1 Runnable Entity Invocation**" and "**5.11.5.4.2 Runnable Entity Termination**".

This functionality has to be configured inside "**DaVinci configurator**" tool. For each runnable under measurement:

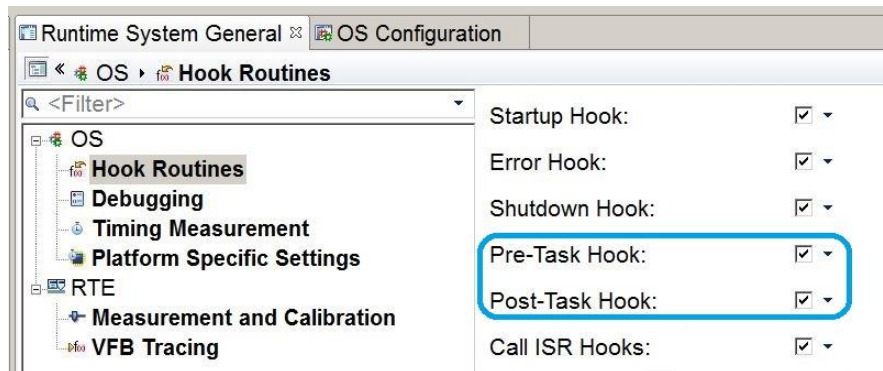
- Open "*Runtime System General*" -> "*RTE*" -> "*VFB Tracing*" -> "*Import VFB Trace Functions Assistant*".
- Select both trace events ("*....\_Start*" and "*....\_Return*").



Implement needed functions respecting recommendation from chapter "5.11.6 Configuration". Inside "*....\_Start*" put call to "**Rtm\_Start(RtmConf\_RtmMeasurementPoint\_<Measurement Name>)**" and inside "*....\_Return*" put call to "**Rtm\_Stop(RtmConf\_RtmMeasurementPoint\_<Measurement Name>)**". Where **<Measurement Name>** is the name of the RTM measurement point configured for this runnable.

#### OS tasks

Time measurement of the OS tasks uses "*PreTaskHook*" and "*PostTaskHook*" functionality described into "**OSEK/VDX Operating System, Version 2.2.3**" (basis of AUTOSAR OS). This functionality has to be configured inside "**DaVinci configurator**" tool.



These settings will enable calls to functions "**void PostTaskHook(void)**" and "**void PreTaskHook(void)**".

Implementation of these functions must contain:

- PreTaskHook:
  - 1) Call to "**void Rtm\_EnterTaskFct(void)**"
  - 2) Call to "**Rtm\_Start(RtmConf\_RtmMeasurementPoint\_<Measurement Name>)**", where **<Measurement Name>** is the name of the RTM measurement point configured for the OS task with ID returned by OS API "**GetTaskID**".
- PostTaskHook:

- 1) Call to "**Rtm\_Stop(RtmConf\_RtmMeasurementPoint\_<Measurement Name>)**", where **<Measurement Name>** is the name of the RTM measurement point configured for the OS task with ID returned by OS API "**GetTaskID**".
- 2) Call to "**void Rtm\_LeaveTaskFct(void)**".

#### 8.14 Display management

Many displays have recommended sequence for starting and stopping the device. Typically such sequences include handling of several pins with respect to some timing constraints. The intended scope of devices include TFTs, DOT matrix displays, LCDs. Remote devices (devices which are part of other ECUs) could also be supported. Description for proper start/stop of the device are given in the device datasheet.

This common use-case raises the need of software entity that handles these activities. Its main functionality is to handle start and stop sequences. The contents that are going to be visualized on the display is functionality that is not targeted by this software entity. The paragraph here includes some constraints and requirements to the design of this software entity:

##### R.DISPMGMT.001

Software architecture shall define software asset that performs device start and stop procedures.

##### R.DISPMGMT.002

The software asset to be defined shall provide the possibility to request starting and stopping of the device on demand.

##### R.DISPMGMT.003

The software asset to be defined shall abstract its users from the actual way of performing start and stop sequence.

##### R.DISPMGMT.004

The software asset to be defined shall be able to perform start/stop sequence for multiple devices.

##### R.DISPMGMT.005

The software asset to be defined shall contain common, device-independent part that can be reused for the various devices to be handled

##### R.DISPMGMT.006

The software asset to be defined shall isolate device specific code. The actual device/s to be controlled shall be chosen by configuration.

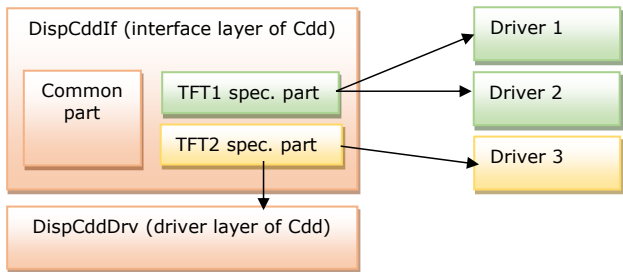
##### 8.14.1 Design decision

The requested functionality will be realized by complex device driver (Cdd), named DispCdd. It will contain manager layer that will be common and not changed. The specifics, introduced by the device that is controlled, will be handled in the underlying "If" and "Drv" layer. For every device, there will be combination of specific "If" part and one or set of drivers that realize the intended functionality.

Cdd driver layer will be responsible for initialization of FPD-Link Converter peripheral. It is going to be used in case the device has to transmit LVDS signal.

The diagram below describes the following use case:

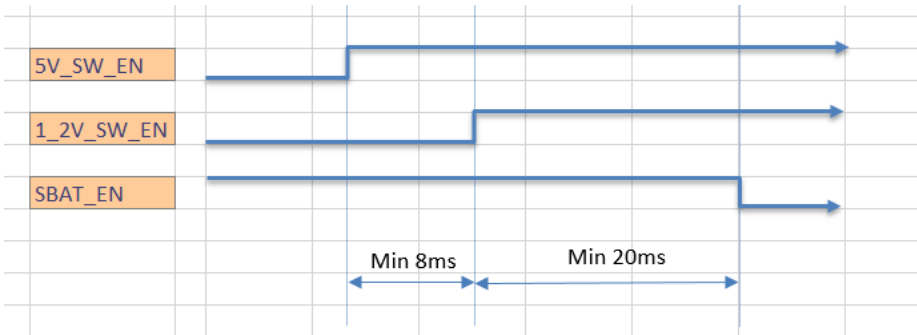
- 2 devices to be controlled
- Both devices use different drivers to perform start/shutdown.
- 1 of them requires FPD



For full description, refer to Turing UML, DispCdd design unit.

8.15 Power On/Off sequence

Power on/off sequence is the process of switching on or off software enabled power supplies. Most commonly such sequences are described in Hardware-Software Interface Specification (HSIS). The special thing about them is that they come with a set of timing requirements. Example of such sequence is the following one:



The following design requirements describe the requested functionality:

**R.PWRSEQ.001**  
Power On sequence shall impact software startup as less as possible

Rationale: Various startup times (wakeup-CAN; wakeup-sound; wakeup-telltale; wakeup-display on) are directly dependent on Power On sequence. Design shall ensure that no extra time is spent on waiting for switching on some power supplies.

**R.PWRSEQ.002**  
Design shall consider the cases when power ON and/ or Power off procedures are not present

Rationale: In most cases there is no specific procedure for Power Off. Additionally, given HW may not require specific Power On seq.

**R.PWRSEQ.003**  
The exact state of Power On sequence shall be provided

Rationale: We don't need information in the sense of power on seq. completed/not completed. We need to know at every moment which power supply is already enabled/switched on. This is needed because, certain functionalities may depend on the power supply that is switched on at the beginning of the Power On sequence. In this case we don't need to wait for the rest of the power supplies. We may run the functionality, right after its power supply is already provided.

**R.PWRSEQ.004**

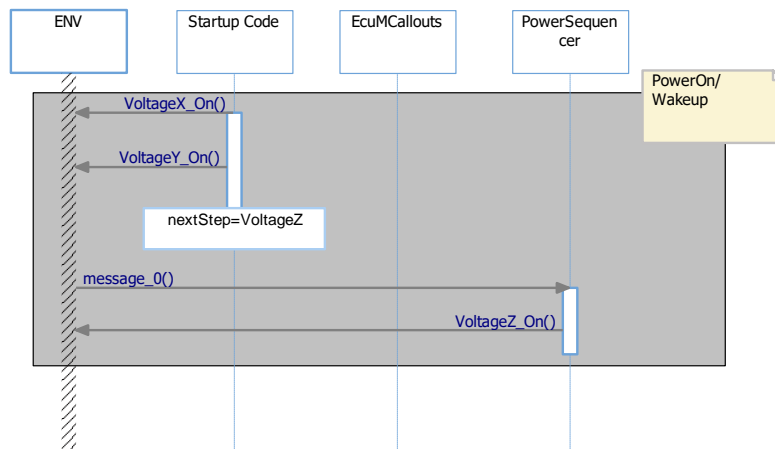
Design shall ensure that power supplies are going to be set again in case of "early wakeup" (device receives a wakeup event on its way to sleep).

Rationale: System shall be kept functional, in case of such situation

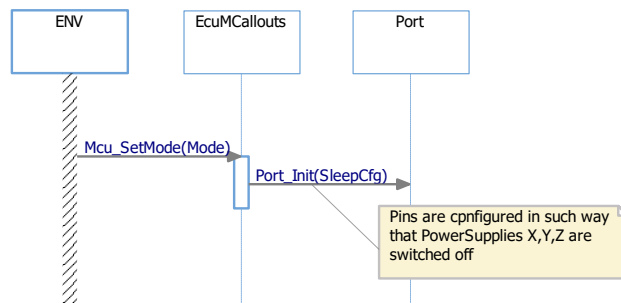
**8.15.1 Design decision**

- Since power on/off seq. is tightly coupled with management of lifecycle components, the most appropriate component to handle this functionality is EcuMExt. Additionally, EcuMExt is notified in case of early wakeup. (R.PWRSEQ.003, R.PWRSEQ.004)
- Startup code is allowed to switch on some or all power supplies, respecting the timing requirements. (R.PWRSEQ.001)
- Even startup code manages some/all power supplies, EcuMExt shall be able to drive them as well. This will be needed in case of early wakeup, when startup code will not be executed (R.PWRSEQ.004)
- If no specific power off sequence is provided by the customer project. The power supplies will be turned off, with Port initialization for sleep mode.

Behavior at startup:



Behavior when going to sleep



## 8.16 Battery management

Information for state of battery is often needed by the projects. Depending on the state, management of some features/resources could be changed. Typical use cases are:

- Stopping resources like backlights, display, telltales, sounds and so on
- change of normal behavior of a feature
- storing of production errors (DTCs)

As a result of the problem statement, 2 main questions exist:

1. How state of battery is going to be determined
2. How actions in case of degraded state are going to be handled

This paragraph explains the Turing's vision about these problems. Design options are listed and design justification is provided.

### 8.16.1 Battery state

#### 8.16.1.1 Design options

##### 8.16.1.1.1 SoHMdl

SoHMdl means application component for determining State of Health for overall device. Important thing here is that SoHMdl is not dedicated to monitoring of under/over voltage. It is more general. Battery state is only one of its factors. Other factor for calculation of State of Health could be some system state coming on CAN or the presence of production mode, where device has very limited functionality. Component main goal is to evaluate all factors and decide which software entities are allowed to run in their normal behavior.

##### 8.16.1.1.2 BattMdl

BattMdl is software component dedicated to monitoring of battery voltage power supply. It has to read current voltage from BSW through standard interface and determine the state of the battery, regarding the specific requirements for the project.

##### 8.16.1.1.3 PowerCdd

PowerCdd is component placed below RTE. Again its main duty is to determine the battery state. The difference with BattMdl is that it may accommodate slightly larger functionality. For example: management of specific power profiles, management of low-level HSIS requirements. Such needs may require handling of interrupts.

#### 8.16.1.2 Design decision

The focus here is battery management and the potential sw assets that are going to be interested in the states of battery. The most appropriate place to handle the change of behavior of output



functions is the application layer. That's why the sw entity computing the state of the battery shall also be part of the application layer. The advantages are:

- Easier synchronization between change of battery state and the affected sw entities (all are SWC => one team will have to manage it (no split of responsibilities))
- Customer logic encapsulated on APP layer (BSW not depended on change in customer SPEC)

Regarding the analysis above, option "[BattMdl](#)" is chosen.

### **8.16.2 Actions in case of degraded mode**

#### **8.16.2.1 Design options**

##### [8.16.2.1.1 CmpLib extension](#)

With this approach CmpLib has to be updated and to include standard degraded modes which can be implemented by components. Component life cycle controller (EcuMExt) will poll battery state and it will transfer configurable set of components in the corresponding degraded mode.

##### [8.16.2.1.2 Client/Server Calls](#)

With this approach the component computing the battery state notifies configurable set of components about change of battery state.

##### [8.16.2.1.3 S/R approach](#)

Components interested in battery state poll variable in RTE, containing information about the battery state.

#### **8.16.2.2 Design decision**

Decision here look somehow easy.

In case of adopting CmpLib/EcuMExt solution, we would need to introduce new state. Introduction of new state would be useful if the component has to change significantly its behavior – for example put all its outputs to 0 and not to operate.

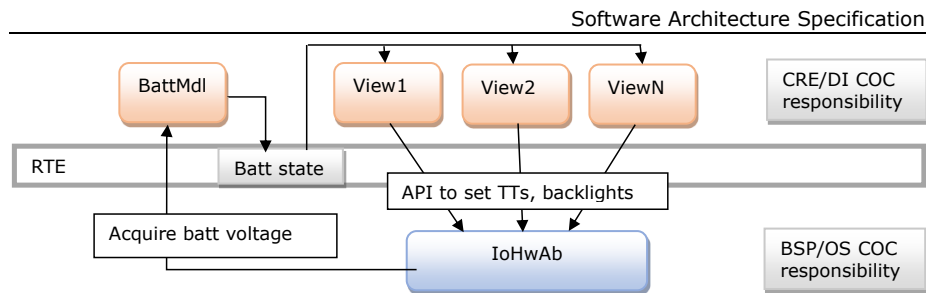
Reality shows we have different use case. We don't want just to stop given resources, we want to change their behavior. For example we don't want all TTs controlled by TtView stopped, we want only some of them or we want to limit backlights to given max percentage (not stop them all)

Client/Server calls have only 1 advantage. They can be faster in terms of reaction. Theoretically this solution shall be faster than S/R approach, since the notified component can apply immediately the change of the internal behavior. This solution has other drawbacks. Portability decreases – the runnables to be notified shall be known when cmp is integrated. If a 3<sup>rd</sup> party Sw shall be notified, we don't know the stereotype of the runnables to be called. Additionally Safety handles much easier S/R communication among partitions. In case of C/S calls crossing the partition boundaries, additional context switches will be required.

As a result of the analysis above, option [S/R approach](#) is chosen.

### **8.16.3 Overall architecture proposal**

As a result of the decision taken above, the following pseudo design is assumed:



## 9 Build

### 9.1 Global defines

The general policy is to reduce as much as possible the usage of global defines in the builds. This is to reduce the overall variants of the software builds, speed up the building process and to reduce the effort for static and dynamic code analysis.

Here are the approved defines for conditional compilation:

- HW-based defines to configure the build for particular HW
- OS-based defines for OSEK configuration
- Autosar required global defines, if any
- Specific defines<sup>(1)</sup>:
  - *UTEST* → Isolates code for unit test purpose only. See R.CODE.STYLE.010<sup>(2)</sup>.
  - *DEBUG* → Enabled in debug builds only. Release build is when this is undefined.
  - *CHECKER* → See R.CODE.STYLE.009<sup>(2)</sup>. This supposed that UTEST is not defined.
  - *MEASUREMENT* → Instruments the code for gathering metrics.

<sup>(1)</sup> These defines are independent from each other.  
Either defined w/o value or undefined **at command line not in the source code**.

<sup>(2)</sup> See porting guideline.

### 9.2 Product variant management with conditional compilation

This article describes the negative impact of using conditional compilation in C/ C++ code to manage product variants.

When projects start, normally they do not have several product variants or have only two, which functionality is well-separated in the following parts: common code reused by both, specific code for the first variant and specific code for the second one.

In order to increase reuse and to minimize the effort for maintenance, the following needs appear along the project lifecycle:

- To have more and more common components with only "light" differences for every variant.
- To use the same code of one non-common component from one variant into another variant with "slight" modification.

One of the techniques that exist in C/C++ is to use conditional compilation in the source and header files to isolate the parts that are not needed when particular variant is built.

Here is a list of problems with their consequences:

#### Naming #defines per product line like PRODUCT\_A, PRODUCT\_B, PRODUCT\_C, etc...

Assume there is PRODUCT\_A and PRODUCT\_B used in one file. Then component is removed from product A and put into a new product C. Source code must be modified, unit test also, without ANY functional change, just because product names are different. Even if it is fixed, later on a new split of components per products might be requested by specification and this useless effort needs to be spent again and again.

Also, trying to handle bigger number of product lines (more than two) via conditional compilation leads to the following issues:

- Source code is very hard to read/analyze (build call graphs, dependency trees, etc.), even by using dedicated tools.
- Source code is hard to debug/trace in a debugger. It is difficult to resolve compilation errors.
- It is difficult to properly separate the code among the different #define sections. Often specific re-factoring of the code is needed, when "moving" code between product lines. This may impact the proper logic, programming style and rules.

Making source code for many product lines leads to have unit tests to be designed to test such, i.e. they also must be for many products. So, the branching of the source code leads to branching in the unit test code - every time the first is modified one needs to check/fix the unit test accordingly. In that case we have manual synchronization of combination of defines used on both sides. There is a high risk unit test not to cover all possible combination of defines, thus many cases won't be tested.

Last but not least, the use of variant specific #defines in a component greatly limits its reusability. It is not known what will be the exact project variants, when the component gets a subject of reuse in the future and how functionality will have to be split between them.

#### Having #define like PRODUCT\_VARIANT with constants PRODUCT\_A, PRODUCT\_B, PRODUCT\_C... or numeric constants, verified in code with expressions

Except the already mentioned problems, here we have in addition the issue with keeping consistency in the enumerated values for the constants. It is entirely manual work. It is possible some of the already used constants for products to be used in files that are not visible to the developer at the moment and he/she will make a new product that duplicates existing one.

Extremely bad practice is to use expressions > >= < <= != like:

```
#if ((PRODUCT_VARIANT <= PRODUCT_B) || (PRODUCT_VARIANT != PRODUCT_C))
...
#endif
```

When new products are added/removed or constants are changed for some reason the enclosed code will be included or not. But the developers cannot know this because such expressions might be everywhere in the code, which means after modification of this type it is needed to verify all source code and all unit tests for regressions. As you can imagine this is not acceptable at all.

#### Branching component's interface with conditional compilation

```
#ifdef PRODUCT_C
struct TPoint
{
    Long x;
    Long y;
}
bool foo ( TPoint & pt );
#else
```

```
    bool foo ( Long x, Long y );  
#endif
```

This is one of the worst things that could be made using conditional compilation. With the increased number of supported products one and the same component has a tendency to maintain more and more interface variants. It is possible the altering of the interface to transform it to something completely different than the original intention in the beginning of the project.

*For example:* Two different methods (by signature) of the interface of the same component could be dedicated to do the same thing in different products. This could be caused by the fact that it is easy to integrate the component in one of the products if signature of the function is modified. As a consequence reusability of the component and its unit test is gone. Also this means redundant effort to test/ maintain duplicated signatures.

#### Branching component's configuration files with conditional compilation

Usually configuration files are generated by code generators. If they need to generate #if in the target code this means they must have the knowledge which data goes to which product variant. I.e. the source code of the generators and/or resource files from which the code will be generated are polluted with product related markers!

Normally, code generators should be independent of the product variants and resources are maintained in a well-formed databases like SQL, XML which have meta-information what is the current set of products and how to extract data for one particular product during one generation session.

#### Mixing product line variants with HW variants of components

```
#if ((defined PRODUCT_C) && (CPU==Dx4))  
    // some code  
#endif  
...  
#if ((defined PRODUCT_A) && (CPU==Dx3))  
    // some code  
#endif
```

There are HW dependent components in the lower layers that are integrated in several product lines of the same project.

The mistake that is observed is to treat HW variant as a product variant. Actually the same HW can be used in several product variants. Also one product variant changes it's HW - VAVE projects are standard practice nowadays. Having a mixture leads to conditional compilation that makes a strong binding between things of two different natures. Developer tend to put product related features under HW related part of code or vice versa. Later on when one needs to port these features to a different HW, or to test them separately, it becomes impossible. At that moment split is inevitable, which is high risk, additional effort, usually under time pressure.

### 9.3 Component level defines

Both xJCI Gen2 and SOSA apply similar approach to splitting the functionality of individual components. SOSA rules excerpt: *"Conditional compilation, used for functional scaling, is done independent of any specific product. For example, if a server has three optional features (A, B, C), conditional compilation expressions would involve using A, B, and C (e.g. #if (FEATURE\_A\_SELECTED) ...code to support feature "A"... #endif). Product oriented conditional compilation (e.g. #if DEW98 ...code for optional feature ...#endif) is never used."*

The decision is: In case of need to split the component's functionality in parts, it is allowed to use conditional compilation. Specific naming rule is applied to avoid collisions:

`<component_name>_<define identifier>`, where `<component_name>` should match the exact name of the component in SIR (capitalized) and `<define identifier>` is user defined token. Example:

LIB\_MEM\_USE\_STD → Component LIB\_Mem is using standard libraries for memory operations.

#### 9.4 Compiler and linker options



CompilerLinkerOptions.xlsx

Important: `--quit_after_warnings` must be enabled for the home-made source. This should be done step by step, because of legacy code, reused in Turing, needs to be purified.

### 10 Testing

TBD: Testing strategies, tools, harnesses.

### 11 Tools

TBD:

UML  
InstancEr  
toAtosar.jar  
TB\_RTE.jar

## 12 Appendix

### 12.1 Trainings

#### 12.1.1 Agenda for Design Champions to explain individual components

For every component/sub-system, which is supposed to be delivered from HLD to the team for DD the following topics have to be explained in the given order:

- Purpose
  - Functionality dedicated to it
  - Role in the system
  - Important use-cases
- Dependencies to other components/sub-systems/frameworks
- Static view
  - Public interfaces – system (framework & OS) and functional ones:
    - What is defined and closed for modification
    - What remains for DD → this is very important to be understood by the trainees.
  - Predefined detailed design elements, if any
    - Functional blocks
    - Private interfaces
  - File structure
  - Tool chain, if any
  - Critical resource (RAM/ FLASH) constraints
- Dynamic view
  - System
    - Life-cycle of the component:
      - Initialization sequences
      - Sleep sequences
      - Wakeup sequences
      - Enabling/disabling
    - Scheduling/calling of the system runnable(s)
    - Usage of system services
      - Framework(s) ones
      - OS ones
  - Functional
    - Clarification of all functional interfaces → when to be called.
    - Normal mode(s)
    - Diagnostics mode
    - End-of-line mode
  - Critical timing constraints, including CPU utilization budget, if any
- Robustness requirements (from HLD) → clarification.
- Safety, if any
- Hints for unit testing and integration

### 12.1.2 Check-list for detailed design completeness

The following questions/hints are not an exhaustive list of all possible things that might be needed to complete the detailed design. They are intended to help not to miss important things.

#### Are public interfaces enough to fulfill the purpose and functionality of the component?

This is the first important question to be answered.

The fact that DC defined the interfaces in HLD, before DD starts, does not mean that those are enough/correct. Usually DC has no chance to read all specifications for that component → they come in portions and might have incomplete/incorrect parts. The responsible for the component need to perform analysis of the provided HLD descriptions against the specifications and to judge whether they are OK or not. In case of issues, meeting with DC to clarify need to be held.

#### Check all requirements from the specification are covered somewhere in the detailed design.

This is for the requirements, dedicated to be implemented in the current sprint.

#### Check that all HLD constraints are respected by the detailed design.

Those that are planned for the current sprint.

Is this the last sprint when it will be possible to achieve the goals (to fit the constraints)?  
Do all optimizations as early as possible → to have more time for test/fix.

#### Is the functionality correctly split among blocks?

Use the design rules, given above in this document.

#### Are individual functional blocks encapsulated?

Having only one purpose, i.e. one reason to change?  
Not depending on things they do not need to?

#### Are individual functional blocks testable?

How exactly they can be tested entirely?  
Are the blocks too complex? Or maybe not complex enough?

#### Error handling and fault handling

Does component distinguish those?  
Not handled, occurring in the component?  
Suppressed within the component?  
Indifferent behavior? → The errors come from outside and are dispatched to other components.  
They might not affect the current component under design.

#### Provisions for the future

Are they needed?  
Are they designed?  
When is the right time to introduce them?

#### Run Autocode generation when detailed design is completed.

This is needed to verify the correctness of the UML syntax for that component. If there are issues in the log, the model has to be fixed. UML syntax is essential for generation of ARXML files for RTE.