VECTOR

Work Guideline

Component Design and Implementation

V2.01.00 | 2016-08-11

Scope and History

This presentation is a Work Guideline (WG).

The contents of this presentation is compulsory for the development of components.

This Work Guideline is **only applicable to MICROSAR 4** development.

This Work Guideline does not apply to CANbedded nor MICROSAR 3.

Slides that contain no binding procedures are marked with the word “informative” on the top right corner.

For existing components please note the defined migration procedure.

Author	Date	Version	Remarks
Jonas Wolf	2014-08-23	0.9	Initial creation
Jonas Wolf	2014-09-23	1.0	Ready for review
Jonas Wolf	2015-07-08	1.3	Move traceability to new WG
Timo Vanoni	2015-11-24	1.4	Add Section “Hardware Software Interface” Correction of BSWMD Traceability
Jonas Wolf	2016-02-03	2.00.00	Release with new template
Markus Schwarz	2016-08-11	2.01.00	Corrections and clarifications; Release

Agenda

- Scope and History
- **Glossary and Abbreviations**
- Design goals
- Component decomposition
- Design patterns
- Naming conventions
- Design documentation
- Coding standard
- Migration procedure

Abbreviations

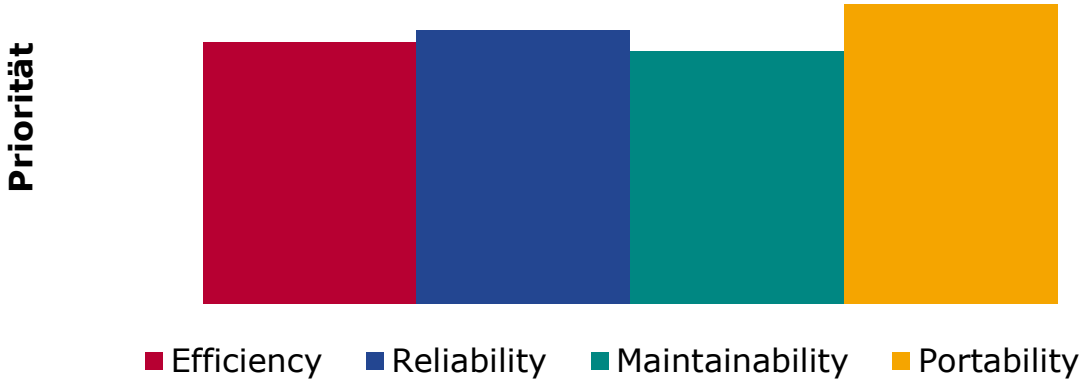
Abbreviation	Description	Example
<MSN> / <Msn>	Module short name in upper and camel case. Note: This has been replaced by <Mip> and <Ma>.	CAN/Can
<MIP> / <Mip>	Module implementation prefix in upper and camel case. Rule: <Ma>[_<vendorId>_<vendorApiInfix>] Note: If the vendorApiInfix has not been defined by the module implementation <Mip> equals to <Ma>	Can_30_Rh850Rscan / CAN_30_RH850RSCAN
<MA> / <Ma>	Module abbreviation in upper and camel case as specified in the AUTOSAR List of Basic Software Modules.	CAN/Can
<SubComp>	Sub-component name. Rule (Regex: [A-Z][A-Za-z0-9]*) - Shall start with an upper case letter - Shall consist out of the following chars: A-Z, a-z, 0-9 - Shall not contain any underscore	Rx, Tx

Prioritization

The following diagrams defines the major design goals for MICROSAR component development.

Functionality is addressed on product level by the feature planning process.
There are no design decisions left for the component developer with respect to functionality.

Safety is addressed separately for product liability reasons.
Safety is mainly addressed by the development process. In case reliability and safety are the same, reliability dominates all other design goals.



The diagram shows the relative weighing of the individual design goals. The higher the bar, the more important is the design goal.

AUTOSAR specification has the higher priority against the rules specified within this WG.

VECTOR

Criteria

- If a component can be split in logical sub-parts they should be decomposed into logical sub-components
  - Note, that a logical sub-component does not necessarily lead to a separate C file.
  - Elements belonging to a logical sub-component should have a unique prefix unless otherwise specified by AUTOSAR.
- If metric thresholds are exceeded a C function should be decomposed into sub-functions
  - The goal is reduction of complexity in order to improve readability and testability.
  - Sub-functions should be declared as <MIP>_LOCAL_INLINE or <MIP>_LOCAL (static).
  - Parameter and plausibility checks of the calling API should not be repeated in a sub-function.

VECTOR

Metrics

- Metric thresholds for C function decomposition

Metric	Threshold
Nesting depth	6
Cyclomatic complexity	20 *
Path count	80
Nesting depth of preprocessor statements	6

- Nesting depth of preprocessor statements is currently not measured.

- Note* Threshold is chosen that high as external APIs must fulfill the API pattern. The API pattern increases the complexity metric without adding “real” complexity, as it is a general pattern.

VECTOR

Files

- Prefix any file of a component with the module implementation prefix (MIP).
- Follow this naming scheme for the applicable files of the component.

File Name	File contents
<Mip>.c	Implementation of the component.
<Mip>.h	Declaration of the component‘s public API visible to other MICROSAR (or SWC, etc.) components.
<Mip>_Int.h	Additional header for component internal API.
<Mip>_Cfg.h	Header for configuration.
<Mip>_Cfg.c	Precompile time configuration.
<Mip>_Lcfg.c	Link time configuration.
<Mip>_PBcfg.c	Post build configuration.
<Mip>_Cbk.h	Declaration of the component‘s callback functions.
<Mip>_Irq.c	Implementation of interrupt service routines.
<Mip>_Irq.h	Declaration of functions/variables for interrupt processing, esp. interrupt service routines.
<Mip>_<SubCmp>.h	Additional header if component is divided into several sub-component.
<Mip>_<SubCmp>.c	Implementation of a sub-component.
<Mip>_<SubCmp>Int.h	Additional header for component internal sub-component API.

VECTOR

Data structures

- Group data that logically belongs to each other in `struct`s.
  - ➔ Maintainability
- Sort attributes of a `struct` in descending order by their data type size.
  - ➔ achieve an preferably good alignment
- Group attributes with the same data type size by their expected access frequency.
  - ➔ increase probability of cache hit

```
struct
{
  uint32 x;
  uint32 y;
  uint32 z;
  uint16 a;
  uint8 b;
} xyzab;
```

Attributes of the `struct` are sorted by their data type size.

VECTOR

Version check

- All c/h files in the implementation package have the same version.
  - This version is defined in the <Mip>.h file (see template for naming convention).
  - This version is checked in the static C file <Mip>.c.
  - For an example refer to the code template.
- Decimal coding of the version number is preferred.
- The coding of the version number shall be documented in the technical reference.
- It is recommended to check also the version of the generated configuration files which depends on the version of the code generator.

VECTOR

Initialization

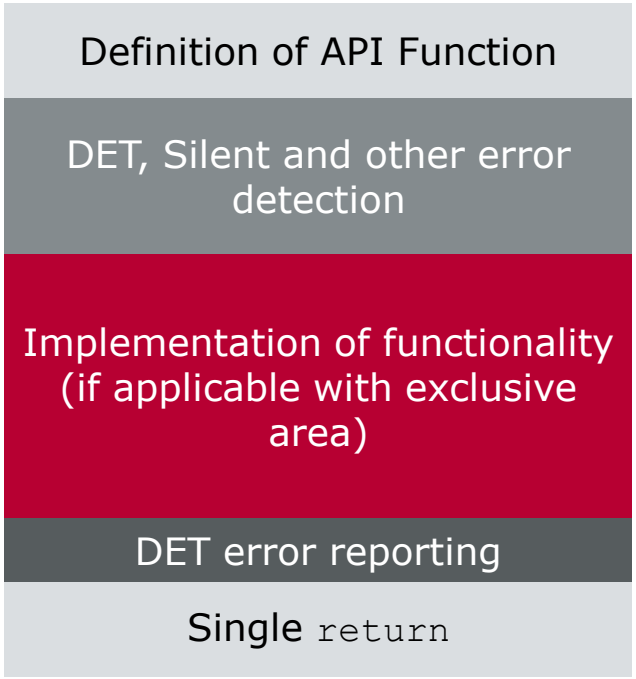
- Create a `<Mip>_InitMemory` function for a component.
- This function only initializes all variables of the component that have an initialization value or mapped to `VAR_INIT` or `ZERO_INIT`.
- Do not initialize variables that are initialized during `<Mip>_Init()`.
- It returns nothing and has no parameters.
- See template for prototype.

- Motivation:
  - Some users have optimized startup code where not all variables are initialized
  - Reduced reliance on build chain parts which could be adapted by the user

API Functions – Pattern

- Follow the this pattern to implement a function of your component wherever possible.
- This pattern is mandatory for all public API functions.
- DET checks shall not be repeated in local functions.

- Crucial points:
  - Single `return` at the end of the function.
  - DET report once at the end of the function.
  - Default return value is `E_NOT_OK`.
  - Functionality is implemented in the `else` path.
    - If additional checks need to be implemented, apply the same `if`, `else if`, `else` pattern with an additional level of indentation.
  - `else` path is (usually) the only place where return value is set to `E_OK`.



API Functions – Example

```
FUNC (Std_ReturnType, COMP_CODE) Comp_APIFunction( CONSTP2VAR (void, AUTOMATIC, COMP_APPL_VAR) a,
                                                   CONST (uint32, AUTOMATIC) b )
{
    Std_ReturnType retVal = E_NOT_OK;
    uint8 errorId = COMP_E_NO_ERROR;
#if (COMP_DEV_ERROR_DETECT == STD_ON)
    if (a == NULL_PTR)
    {
        errorId = COMP_E_PARAM_POINTER;
    }
    else if (b > COMP_MAX_VALUE)
    {
        errorId = COMP_E_PARAM;
    }
    else
#endif
    {
        uint8_least i;
        COMP_ENTER_CRITICAL( COMP_EXCLUSIVE_AREA_0 );
        for (i = 0; i < COMP_MAX_VALUE; i++)
        {
            a[i][b] = 1u;
        }
        COMP_LEAVE_CRITICAL( COMP_EXCLUSIVE_AREA_0 );
        retVal = E_OK;
    }
#if (COMP_DEV_ERROR_REPORT == STD_ON)
    if (errorId != COMP_E_NO_ERROR)
    {
        (void)Det_ReportError(COMP_MODULE_ID, COMP_INST_ID, COMP_API_ID, errorId);
    }
#else
    COMP_DUMMY_STATEMENT(errorId);
#endif
    return retVal;
}
```


API Functions – Example 2

```
FUNC (Std_ReturnType, COMP_CODE) Comp_APIFunction( CONSTP2VAR (void, AUTOMATIC, COMP_APPL_VAR) a,
                                                   CONST (uint32, AUTOMATIC) b )
{
    Std_ReturnType retVal = E_NOT_OK;
    uint8 errorId = COMP_E_NO_ERROR;
#if (COMP_DEV_ERROR_DETECT == STD_ON)
    if (a == NULL_PTR)
    {
        errorId = COMP_E_PARAM_POINTER;
    }
    else if (b > COMP_MAX_VALUE)
    {
        errorId = COMP_E_PARAM;
    }
    else
#endif
    {
        COMP_ENTER_CRITICAL( COMP_EXCLUSIVE_AREA_0 );
        if (CheckThatNeedsCriticalSection() == FALSE)
        {
            errorId = COMP_E_INVALID_STATE;
        }
        else if (AnotherCheckThatNeedsCriticalSection() == FALSE)
        {
            errorId = COMP_E_INVALID_STATE;
        }
        else
        {
            uint8_least i;
            for (i = 0; i < COMP_MAX_VALUE; i++)
            {
                a[i][b] = 1u;
            }
            retVal = E_OK;
        }
        COMP_LEAVE_CRITICAL( COMP_EXCLUSIVE_AREA_0 );
    }
#if (COMP_DEV_ERROR_REPORT == STD_ON)
    if (errorId != COMP_E_NO_ERROR)
    {
        (void)Det_ReportError(COMP_MODULE_ID, COMP_INST_ID, COMP_API_ID, errorId);
    }
#else
     COMP_DUMMY_STATEMENT(errorId);
#endif
    return retVal;
}
```

VECTOR

General Defines

- The General Define Block in each `<Mip>_Cfg.h` provides
  - `#define <MIP>_DUMMY_STATEMENT`
    - based on parameter of EcuC module
- Each component
  - `#define <MIP>_DEV_ERROR_DETECT` and
  - `#define <MIP>_DEV_ERROR_REPORT`
    - based on own parameter <Mip>DevErrorDetect and global parameter for SafeBsw within EcuC module.
    - Example

```
#define COM_DEV_ERROR_DETECT
<%=F.formatBool(comData.getEcucGeneralLegacy().getEcuCSafeBswChecks().getValue() ||
comData.getComGeneral().getComConfigurationUseDet().getValue()).asStdOnOff()%>
/**< <%=comData.getEcucGeneralLegacy().getEcuCSafeBswChecks().getObjectLink()%> ||
<%=comData.getComGeneral().getComConfigurationUseDet().getObjectLink()%> */


#define COM_DEV_ERROR_REPORT
<%=comData.getComGeneral().getComConfigurationUseDet().formatValue().asStdOnOff()%>
/**< <%=comData.getComGeneral().getComConfigurationUseDet().getObjectLink()%> */
```

VECTOR

Decisions

- Formulate decisions that have an `if` and an `else` path always positive.
  - Example:

```
if (a == TRUE)
{
    /* Do A. */
}
else
{
    /* Do B. */
}
```
Good

```
if (a == FALSE)
{
    /* Do B. */
}
else
{
    /* Do A. */
}
```
Worse

```
if (a != TRUE)
{
    /* Do B. */
}
else
{
    /* Do A. */
}
```
Worst

- If an `else` is not necessary for a decision, do not introduce an `else` path artificially to comply with the above rule.

- Name the expression `a` in a positive way, too.
  - Example:
    - `exists` in stead of `missing`
    - `Initialized` instead of `uninitialized`

VECTOR

Pointer arithmetic

- Use indexed access to arrays only.
- Do not use other forms of pointer arithmetic.

Allowed

```
uint32 a[10];
uint32 i = 0;
uint32 *p;

a[i] = 5u;
i++;
a[i] = 6u;

p = &a[7];
```

Not allowed

```
uint32 a[10];
uint32 *p;
uint32 offset;

offset = 5u;

p = *(a + offset);
```

VECTOR

Comparison to constants

- Put constants on the right side of the decision to support the natural reading direction.
  - ➔ Potential faults are detected by MISRA and/or compiler check
- Example:

```
if (a == TRUE)
```
Good

```
if (TRUE == a)
```
Worse

VECTOR

Loops

- Use incremental `for` loops for iterating loops.
  - Incremental counting is natural ➔Maintainability

- Use `<type>_least` data types for loop variables.
  - Use the data type for `<type>` that has the smallest size and covers the complete needed value range.
  - ➔ Efficiency on different platforms

- Example:

```
uint8_least i;
for (i = 0; i < COMP_MAX_VALUE; i++)
{
    /* Do things in loop. */
}
```

Nested loops

```
…

for (i = 0u; i < COMP_MAX_I_VALUE; i++)
{
    for (j = 0u; j < COMP_MAX_J_VALUE; j++)
    {
        /* Do work in loop. */
        if ( <Condition to leave both loops> )
        {
            break;
        }
    }
    if ( <Condition to leave both loops> )
    {
        break;
    }
}

…
```

Leave nested loops via duplicated `break`.

The compiler generates efficient code because the termination condition is checked only once.

Readability is improved. At the same time there is still flexibility to insert additional statements before the second `break`.

VECTOR

Macros

- All objects visible at file scope are prefixed with a unique prefix.

Identifier	Pattern	Example
Service IDs for DET reporting	<MIP>_SID_<SID>	ECUM_SID_MAINFUNCTION COM_SID_INIT
Error Ids for DET reporting	<MIP>_E_<ERRORID> <ERRORID> can be e.g. > NO_ERROR (used to check if no error occurred), > PARAM_CONFIG (API service <Mip>_Init() called with wrong parameter), > PARAM_POINTER (API service called with invalid pointer parameter (NULL)), > PARAM_<x> > UNINIT (API service used without module initialization), > ALREADY_INITIALIZED (The service <Mip>_Init() is called while the module is already initialized)	CANIF_E_PARAM_CONFIG
Configuration Variant Macros	<MIP>_CFG_<VARIANT>	NVM_CFG_
DET Error Detection	<MIP>_DEV_ERROR_DETECT	
DET Error Reporting	<MIP>_DEV_ERROR_REPORT	
Inline Macros	<Mip>_<MacroName>	Com_DecRepCnt
Exclusive Areas	<MIP>_EXCLUSIVE_AREA_<ID>	CANIF_EXCLUSIVE_AREA_0

VECTOR

Variables and Types

- Use the following naming conventions for variables and types.

Identifier	Pattern	Example
Global variables (external)	<Mip>_<VariableName>	CanIf_ControllerConfig
Global variables (static)	<Mip>_<VariableName>	EcuM_LastShutdownMode
Local variables	<variableName>	loopCount

Identifier	Pattern	Example
Type name	<Mip>_<TypeName>Type	Fr_ChannelType
Enum	Unnamed (if typedefed)	typedef enum {…} Fr_EnumType;
Struct	Unnamed (if typedefed)	typedef struct {…} Fr_SampleStructType;
Enum literals	<MIP>_<NAME>	FR_UNINIT
Struct Member	<Name> (no prefix)	… uint8 StructMember;

VECTOR

Overview

- The component design documentation consist of
  - the component abstract design (CAD)
  - the component detail design (CDD)

Kind	Aspect	Methodology
CAD	Component structure	Enterprise Architect
CDD	Internal behavior of each function	doxygen
	Data structures	doxygen
CDD	Configuration	Information in BSWMD files.

VECTOR

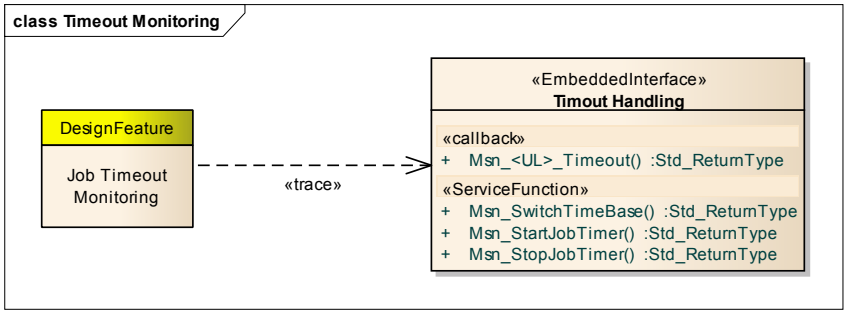
CAD

- The CAD describes
  - Functional aspects
    - the state machine(s) of the component (if applicable)
    - the important sequences of the component
  - Logical aspects
    - the sub-components of the component (if applicable)
  - Structural aspects
  - Conceptual aspects

- The CAD is documented using Enterprise Architect.
- The CAD is based on a template which considers the views above and limits the EA toolbox accordingly.
- CAD items are: DesignFeatures, Concepts, StateMachines, LogicalUnits

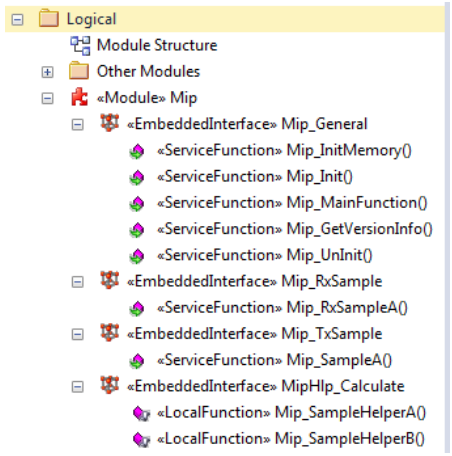
CAD - Functional

- Use DesignFeatures to document the solutions of the component functionalities.
- Set the SafetyLevel for each DesignFeature.
- DesignFeatures and Concepts can be traced to Embedded Interfaces or other elements in the CAD.
  - Use a Dependency with stereotype “trace” to show that a DesignFeature or Concept or parts of it are realized in the target of the relation.
    - Tracing to an Embedded Interface is equivalent to using the“\trace” command in the Doxygen comments of the contained functions.
    - If only a subset of the functions of the Embedded Interface is related, for each relevant function, add its name in a separate line to the notes of the relation.
  - Use a Dependency without keyword to show that a DesignFeature or Concept is related to another DesignFeature.

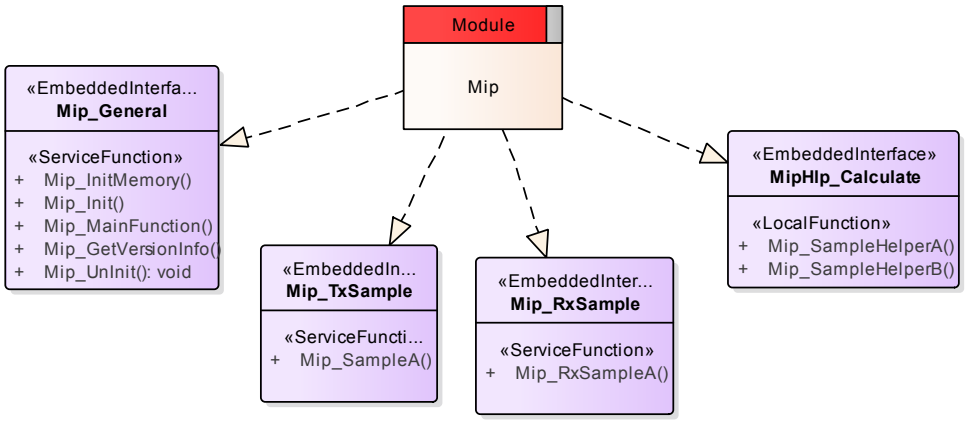


Interfaces to other components

In the following example diagram a component consisting with sub-components is used:

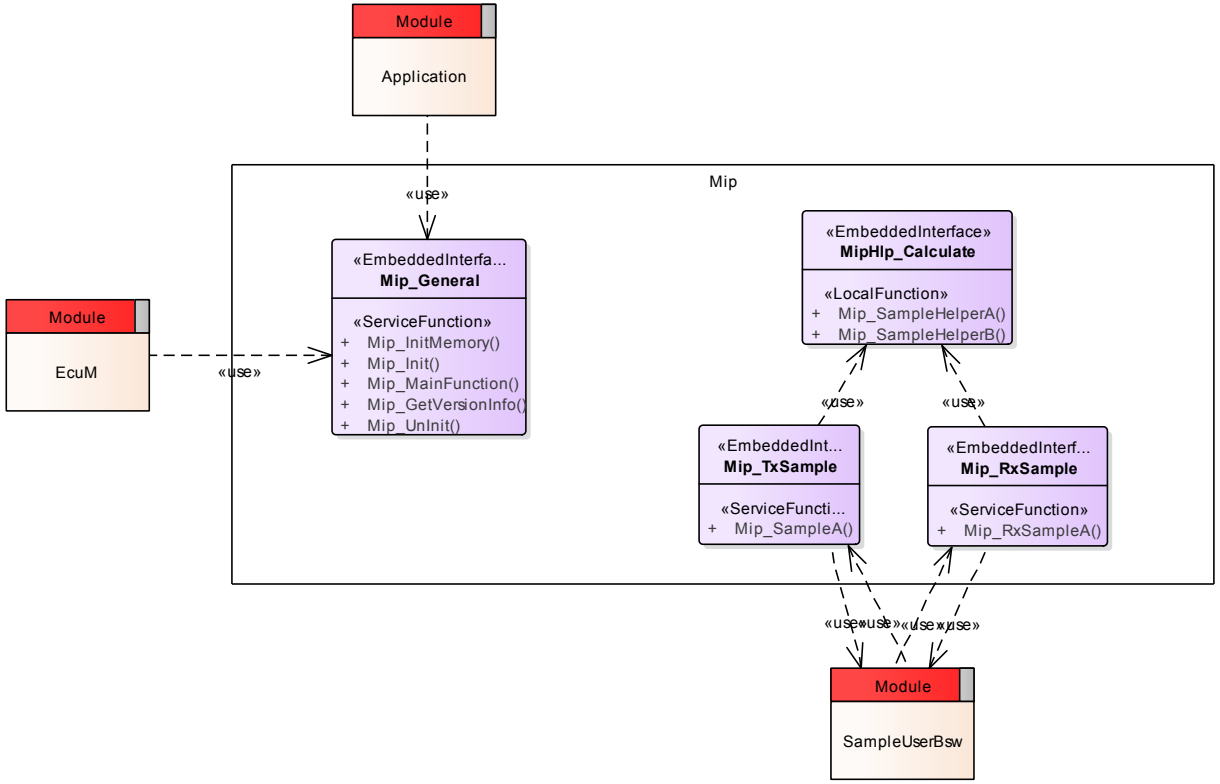


- Use the stereotype *EmbeddedInterface* and a *realizes* relation to model externally visible interfaces of your component:



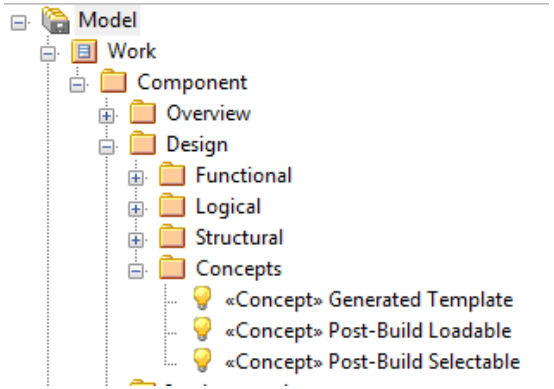
Sub-components of the component

- Use a class diagram and an *implements* relation to show the sub-components and their interfaces:
- Use only the name for API functions.
- There is no need to document parameters and a description in Enterprise Architect. This information is documented in doxygen.
- Textually describe the functionality of the component its *Notes* window.
- Set the SafetyLevel for each function.



Concepts

- Document important concepts of the component on an abstract level
  - Example: post-build



  - In description add the following information
    - \trace reference to general architecture concept if applicable
    - Brief list of properties on an abstract level, details are specified in the CDD (BSWMD file in case of post-build)
      - Example for post-build:
        - Adding of signals
        - Modification of signal parameters
        - …
    - \trace reference to related usage scenario based CREQ (if applicable)

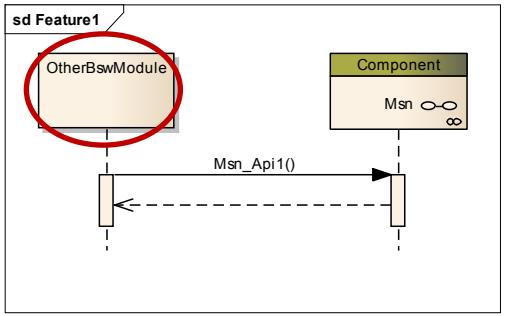
VECTOR

Important sequences

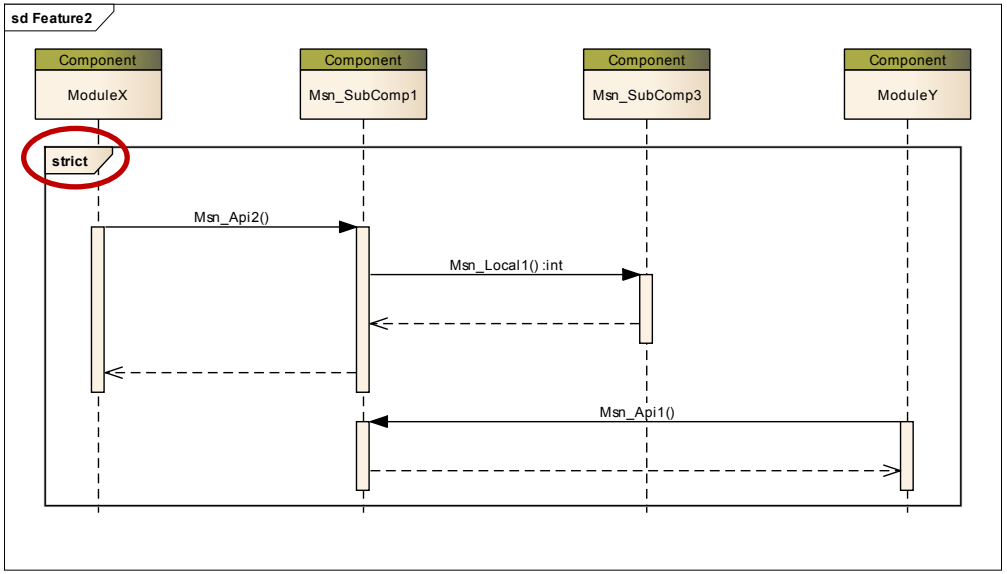
1. Document the following sequences:
   1. External interfaces of the component where the **order of the sequence** is important.
   2. External interfaces of the component where **two or more other components** are involved.
   3. External interfaces of the component where **asynchronous behavior** is exposed.
   4. External interfaces of the component where **synchronous behavior** is exposed and where based on experience **unusual latency** has to be expected e.g. through hardware access or complex algorithms.
   5. Sequences that are a **deviation from the AUTOSAR** standard.
   6. **Failures** that based on the **safety analysis** are marked as „red" and require more than a parameter check within the function.
2. Do **not** document the following sequences:
   1. External interfaces of the component where just one function is involved (e.g. GetVersion).

Presentation of important sequences – 1

- Use the following presentation if the calling component is unknown:



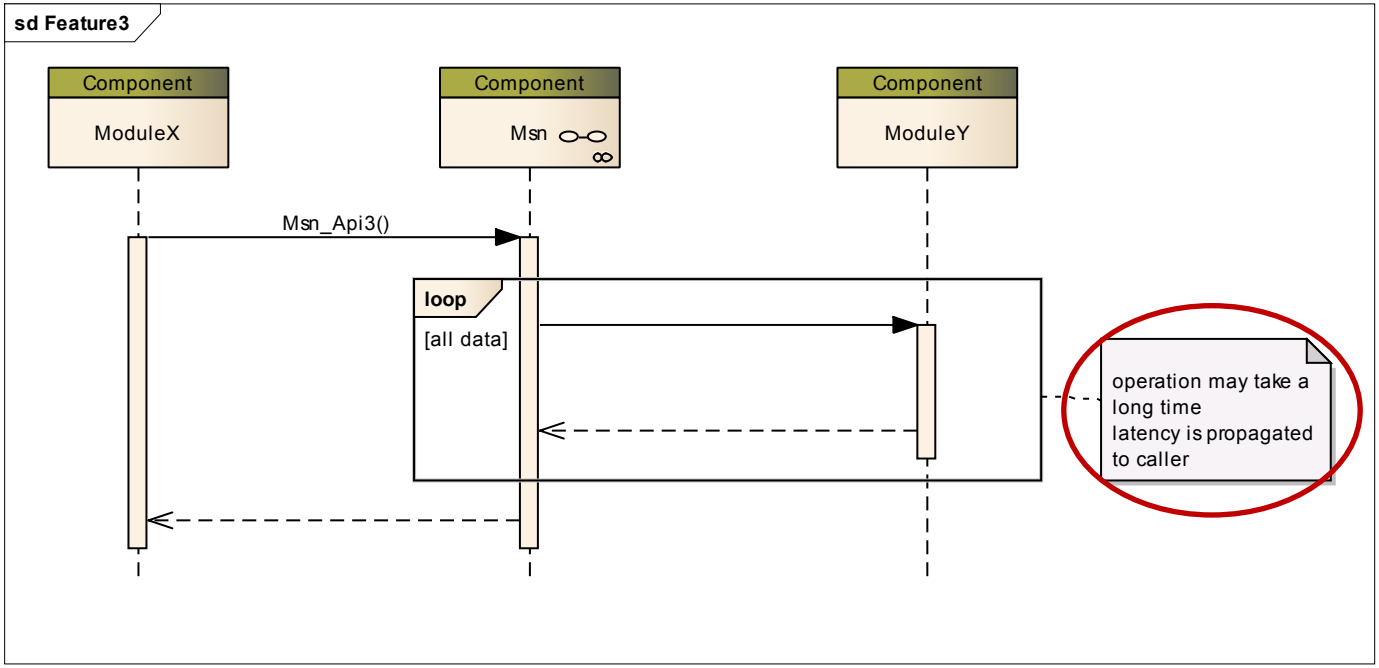
- Use the following presentation if the order of sequence is important:



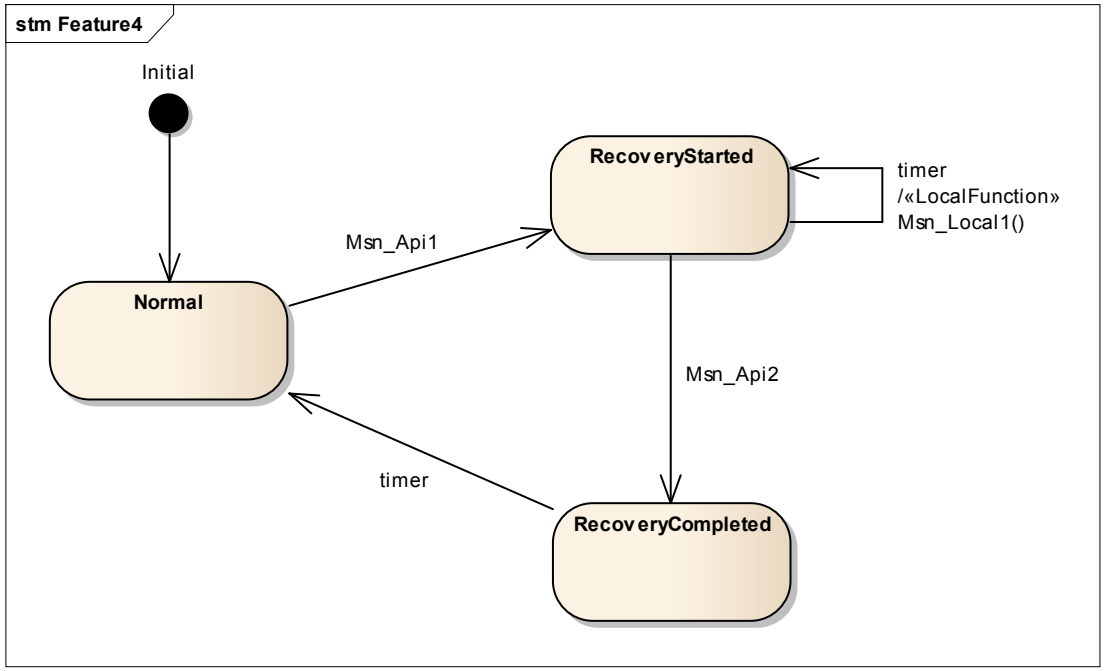
VECTOR

Presentation of important sequences – 2

- Use the following presentation if unusual latency is expected:



State machines



- Use state machine diagrams for the representation of state machines.
- Nesting of state machines is allowed.
- Model behavior of state machines as actions at transitions.

VECTOR

CDD

- Use the sequence `/*!` to define a comment interpreted by doxygen.
- Use the backslash `\` for doxygen commands.

- Place the CDD in the static source files.
- Place the CDD for generated functions in a static source file and:
  - refer from the CDD to the generated function by using the doxygen command `\fn <FunctionName>` or
  - refer from the generated function to the CDD by making the CDD identifiable (“e.g. CDD for pattern <identifier>”) and placing a comment over the generated function that refers to this identifier (e.g. “implements pattern <identifier>”).
- If the component has no static source file then document the CDD in the generator (e.g. within the templates).
  - Output the externally visible CDD (e.g. function declarations) but **not** the contents of the `\internal` blocks.

VECTOR

Function declaration – 1

- Document **every** function declaration with the following doxygen commands.
- This also applies to generated function code.

doxygen Command	Description	Example
brief	Document in one, short sentence the functionality of the function. You can use „Description" of SWS as template to sketch the functionality.	▸ Initializes component.
details	Provide more details about the functionality.	
param[in|out|in,out]	Document every parameter with one tag per parameter, use the [in], [out] or [in,out] attributes. Document the value range if not the complete value range of the type is defined by an enumeration. Document assumptions of the parameter, e.g. that the parameter has to be checked by the caller. In case of a pointer parameter, document the constraints for the referenced object (e.g. the number of elements in the buffer). Omit if the function has no parameter.	▸ param[in] Id of the PDU. ▸ param[out] Pointer to an object where to store the version information. Parameter must not be NULL. ▸ param[out] Pointer to an uint8-array with 8 elements where the data is copied to.
return	Describe the functions return value. Omit if the function returns void.	▸ E_OK if status was updated, otherwise E_NOT_OK.
context	Document the calling context by using one of the following identifiers: TASK|ISR1|ISR2|ERRHOOK|PRETHOOK|POSTTHOOK|STARTHOOK|SHUTHOOK|ALARMHOOK|PROTHOOK|ANY Select TASK for <Mip>_Init() functions. ISR1 can only be used if no OS API functions are used.	▸ TASK|ISR2 ▸ ANY ▸ TASK
reentrant	Document reentrancy of function using TRUE or FALSE. If reentrancy depends on other factors, e.g. handles, add a comment after TRUE. Select FALSE only if function is not reentrant at all.	▸ TRUE ▸ TRUE for different handles ▸ FALSE
synchronous	Document if the service is performed synchronously to the function call using either TRUE or FALSE. This only applies to public services provided by the component.	▸ TRUE
config	Describe configuration variant (using the pre-processor switch) in which the function is available. This only applies to public functions. Do not describe configuration variants that only influence the behavior of the function. This is done at the function definition. Omit if the function is independent of the configuration.	▸ VERSION_API_ENABLED
pre	State the pre-condition that has to be fulfilled before calling the function. If the pre-condition is not fulfilled, the function will fail. Omit "Module has to be initialized" or pre-conditions that are already described (e.g. parameter ranges). Use a dash ("-") if there is no additional pre-condition.	▸ Interrupts are disabled. ▸ Module is in state NORMAL.

VECTOR

Function declaration – 2

- Document **every** function declaration with the following doxygen commands.
- This also applies to generated function code.

doxygen Command	Description	Example
trace <CREQ|SI|DSGN>	Document the requirement why this function is needed. You may use one tag per requirement or one tag for all requirements (comma separated). Refer to "WG Traceability" for details. Omit if there is no trace source.	▸ \trace CREQ_1234 ▸ \trace SPEC_5678
trace <SI>	Describe deviations to the AUTOSAR SWS. Deviation implies that a function is implemented in a different manner. Do not use deviations to describe unimplemented functions. Refer to "WG Traceability" for details. Omit if there is no trace source.	▸ \trace SPEC_12345 parameter type changed to create consistency to SPEC_23423.

- Also refer to https://vglobpessvn1.vg.vector.int/svn/Products/MICROSAR4/ProductFeatures/PF4_SafeBSW/trunk/90_ProjectManagement/30_Projekt_2_Process/Komponentenentwurf/BestPractise_DetailDesign.docm

VECTOR

Function definition

- Document the implementation at the function definition.

doxygen Command	Description
internal	Start of description about internal implementation details and concepts, e.g. usage of data structures, algorithms, design decisions. This is internal information which is not intended for the technical reference.
endinternal	End of “internal”

- Motivation: A review should be able to identify if there is undocumented behavior within the function.
- Do not try to rewrite the function using pseudo code.
- Use the term “check” for simplified description of DET checks.
- Only document simple getter/setter functions at the function declaration. Only add `\internal` documentation if more than setting or getting a value is performed.
- For initialization functions, document the groups of variables that are set, e.g. “all”, or only “state variables”.
  - Initialization values do not need to be documented.

Function definition – Example 1

```
1095
1096 #if (RAMTST_RUNPARTIALTEST_API == STD_ON)
1097 /*****************************************************************************************************************
1098  *  RamTst_RunPartialTest()
1099  *****************************************************************************************************************/
1109 FUNC(void, RAMTST_CODE) RamTst_RunPartialTest(RamTst_NumberOfBlocksType BlockId)
1110 {
1111     RamTst_ExecutionStatusType TempExecutionStatus;
1112     P2CONST(RamTst_BlockParamsConfigType, AUTOMATIC, RAMTST_CONST) CurrentBlock_pt;
1113
1114     if(RAMTST_EXECUTION_UNINIT == RamTst_CurrentExecutionStatus_t)
1115     {
1116         /* The current execution is RAMTST_EXECUTION_UNINIT? */
1117         RamTst_DetReportError(RAMTST_SID_RUN_PARTIAL_TEST, RAMTST_E_UNINIT);
1118     }
1119     else if(RAMTST_EXECUTION_STOPPED != RamTst_CurrentExecutionStatus_t && RAMTST_EXECUTION_SUSPENDED != RamTst_CurrentExecutionStatus_t)
1120     {
1121         /* The current execution status is not RAMTST_EXECUTION_STOPPED and */
1122         /* the current execution status is not RAMTST_EXECUTION_SUSPENDED */
1123         RamTst_DetReportError(RAMTST_SID_RUN_PARTIAL_TEST, RAMTST_E_STATUS_FAILURE);
1124     }
1125     else if(RamTst_Config_t.RamTst_AlgParams_at[RamTst_AlgorithmParameterId_t].RamTst_NumberOfBlocks_t <= BlockId)
1126     {
1127         /* The current execution status is not RAMTST_EXECUTION_SUSPENDED */
1128         RamTst_DetReportError(RAMTST_SID_RUN_PARTIAL_TEST, RAMTST_E_OUT_OF_RANGE);
1129     }
1130     else
1131     {
1132         /* ===== Critical Section Start ===== */
1133         RamTst_EnterCritical();
1134
1135         /* Save the current execution status. */
1136         TempExecutionStatus = RamTst_CurrentExecutionStatus_t;
1137         /* Set the execution status to RAMTST_EXECUTION_RUNNING */
1138         RamTst_CurrentExecutionStatus_t = RAMTST_EXECUTION_RUNNING;
1139
1140         CurrentBlock_pt = &RamTst_Config_t.RamTst_AlgParams_at[RamTst_AlgorithmParameterId_t].RamTst_ConfigBlocks_at[BlockId];
1141
1142         RamTst_PerBlockTestResult_at[BlockId]
1143             = RamTst_AlgorithmFuncTable[RamTst_Config_t.RamTst_AlgParams_at[RamTst_AlgorithmParameterId_t].RamTst_Algorithm_t](
1144                     CurrentBlock_pt->RamTst_StartAddress_pu32,
1145                     CurrentBlock_pt->RamTst_StopAddress_pu32,
1146                     CurrentBlock_pt->RamTst_FillPattern_u32,
1147                     CurrentBlock_pt->RamTst_TestPolicy_u8);
1148
1149         /* Check the test result for the specific block. */
1150         if(RAMTST_RESULT_NOT_OK == RamTst_PerBlockTestResult_at[BlockId])
1151         {
1152             /* The test result is not ok. */
1153             RamTst_OverallTestResult_t = RAMTST_RESULT_NOT_OK;
1154             RamTst_DemReportError(RamTst_RamFailure, DEM_EVENT_STATUS_FAILED);
1155         }
1156
1157         /* Toggle the current status back to the status before the call. */
1158         RamTst_CurrentExecutionStatus_t = TempExecutionStatus;
1159
1160         /* ===== Critical Section End ===== */
1161         RamTst_ExitCritical();
1162     }
1163
1164     return;
1165
1166 } /* RamTst_RunPartialTest() */ /* PRQA S 2006 */ /* MD_RamTst_2006 */
1167 #endif  /* (RAMTST_RUNPARTIALTEST_API == STD_ON) */
1168
```

VECTOR

Function definition – Example 1

```
/*!
 * \internal
 * - #10 Check if component is initialized.
 * - #20 Check if test is neither stopped nor suspended.
 * - #30 Check if BlockId is within number of blocks of current algorithm configuration.
 * - #40 Enter critical section (Reason: current execution status may not be changed
 *       elsewhere, RAM test must be executed with locked interrupts).
 * - #50 Temporarily save the current execution status.
 * - #60 Call RAM test algorithm.
 * - #70 If RAM test failed
 *   - #80 Set overall result to failed.
 *   - #90 Report failure of test to DEM.
 * - #100 Restore current execution status.
 * - #110 Leave critical section.
 * \endinternal
 */
FUNC(void, RAMTST_CODE) RamTst_RunPartialTest(RamTst_NumberOfBlocksType BlockId)
```

- Use indentation consistently with your code. DET checks are abbreviated on the same level as the “else”-path.
- If only parameters are checked, the term “`Check plausibility of input parameters`” can be used for simplification.

VECTOR

Language

- Use a simple structure for every sentence: subject – predicate – object.
- Use present tense, even if it is done in future.
- Use indicative form of the verb.
- Use active voice, instead of passive voice.
- Use short sentences.
- Use only complete sentences.
- Make one statement per sentence.
- Use at most one dependent clause.
- Put conditional or final clauses at the end of the sentence.
- Use positive expressions.
- Use the identical terms for the same things.

VECTOR

Wording of descriptions

```
/*!
 * \brief <text>
 * …
 */
void function(void);
```

```
/*! <text> */
type variable;
```

Type	Guideline for text	Example
Function	Start the description of the function with a verb.	Returns the amount of free entries in the list. uint32 List_GetFreeEntries();
Variable, Constant	Start the description of the variable or constant with a verb or a substantive. Describe the value range if it is not already defined by the type.	Indicates whether the A/D conversion is enabled. bool Adc_ConversionEnabled; Number of active A/D conversion channels. uint32 Adc_NumberOfActiveChannels; Current state of the module state machine (Valid values: MIP_STATE_*). uint8 Mip_CurrentState;

VECTOR

Exclusive Areas

- Document all exclusive areas in a doxygen comment block at the end of the module header file.
- It is assumed that a protected resource is only accessed within its corresponding exclusive area. Hence, it is only required to document deviations of this assumption (e.g. when the critical area is not used) in the detailed design of a function.
- Use all of the following commands to describe an exclusive area.

doxygen Command	Description	Example
exclusivearea	Identifier of the exclusive area followed by its purpose in the next line(s).	▸ COMP_EXCLUSIVE_AREA_1 Ensures consistency while modifying the interrupt counter.
protects	Describe the resource that is protected by this exclusive area.	▸ CanNm_TxMessageData ▸ CAN Peripheral ▸ Interrupt Register
usedin	API functions (i.e. root of the call tree within the BSW module) that use this exclusive area.	
exclude	Describe functions / events that are excluded from this exclusive area. If it can be ensured that these functions / events cannot interrupt the exclusive area, no locking mechanism is required.	▸ CanNm_MainFunction, CanNm_RequestBusSynchronization
length	Provide a qualification about the expected length. Use one of the following values followed by an optional description: SHORT, MEDIUM, LONG.	▸ LONG This exclusive area covers calls to several sub-functions.
endexclusivearea	Specifies the end of an exclusive area documentation.	

VECTOR

Exclusive Areas - Qualification of Length

The length of an exclusive area is only a qualitative value based on the expert's estimation and assumption. It is neither calculated nor measured. This slide provides a guide to estimate the length.

- The expected length of an exclusive area is its estimated assumed worst-case.
- When a loop is contained in an exclusive area it's runtime is estimated as N*<number of statements within loop> with N as the expected maximum amount of loop iterations.
- A (inline) function call within an exclusive area is counted as the number of statements within the function.

Value	Description	Example of protected functionality
SHORT	Only a few (< ca. 30) statements with a constant maximum runtime.	▸ Read-Modify-Write operations ▸ Short copy operation ▸ Access to protected hardware registers. ▸ State variable handling.
MEDIUM	More than a few (> ca. 30) statements with a constant maximum runtime and a low call-tree (<=1).	▸ Initialization of large RAM arrays ▸ Access to large structures
LONG	A lot of statements (> ca. 100), deep call-tree (depth > 1), complex code (cyclomatic complexity > 10) or an undefined maximum runtime.	▸ Calls to other BSW modules ▸ Calls to the same BSW module with a large call tree. ▸ Complex code. ▸ Loop that waits for external events such as changing hardware flags.

VECTOR

Exclusive Areas - Examples

```
/*!
 * \exclusivearea CAN_EXCLUSIVE_AREA_1
 * Ensures consistency while modifying the interrupt counter and CAN interrupt lock registers.
 * \protects canCanInterruptCounter, CAN interrupt registers
 * \usedin Can_EnableControllerInterrupts, Can_DisableControllerInterrupts
 * \exclude All functions provided by Can.
 * \length SHORT The interrupt registers and a counter variable are modified.
 * \endexclusivearea
 *
 * \exclusivearea CAN_EXCLUSIVE_AREA_4
 * Ensures consistent entries when writing to the RxQueue.
 * \protects canRxQueueBuffer, canRxQueueInfo (write)
 * \usedin Can_SetControllerMode, BusOff Interrupt
 * \exclude Rx Interrupt, Can_MainFunction_Receive
 * \length SHORT The CAN frame is copied to the RX queue.
 * \endexclusivearea
 */

/*!
 * \exclusivearea CANNM_EXCLUSIVE_AREA_1
 * Protects the global message data from being modified while copying to the SDU.
 * \protects CanNm_TxMessageData (read)
 * \usedin CanNm_MainFunction, CanNm_RequestBusSynchronization
 * \exclude CanNm_SetUserData, CanNm_SetSleepReadyBit
 * \length SHORT At max 8 bytes are copied within this exclusive area.
 * \endexclusivearea
 */
```

VECTOR

Hardware Software Interface – General

- If a module accesses internal or external hardware, place a comment with an internal block with the following content at the start of the hardware specific register definitions:

Section	Description	Example
Hardware manuals	List all hardware manuals that are considered for this component. This also includes the hardware's safety manuals, if applicable. The hardware manual is identified by its title and version or a unique document identifier if applicable. The relevant part in the manual is identified by the chapter, page numbers or by naming the peripheral if its name is unique.	▸ R01UH0076ED200 Rev.2.00, FCN ▸ TC24x Target Specification V2.0 2012-12, Chaper 23
Errata sheets	List all errata sheets that are considered for this component. The errata sheet is identified by its title and version or a unique document identifier if applicable. If only a part of the errata sheet is relevant the part is identified by e.g. the section number, the title or the page number(s).	▸ TC1796 Errata Sheet BC Rel 1.7, MultiCAN_TC.023-036 ▸ TN-RH8-S001A/E Rev. 1.00
Access mechanism	Describe how the hardware is accessed.	▸ Memory mapped registers ▸ SPI
Used registers	Refer to the register definitions within the code.	▸ See definitions below
Hardware features related to independence or partitioning	Describe if hardware features related to independence or partitioning of software are used. Use a dash ("-") if not applicable.	▸ Usage of MPU.
Operating modes	Describe what operation modes are used and their mapping to the software. Refer to the CAD if it is already sufficiently described there. Use a dash ("-") if not applicable.	▸ Software STOP : INIT1 ▸ Refer to the states specified in the CAD.
Hardware diagnostics	Describe how hardware diagnostics is performed and how it is implemented. If hardware features are used for the diagnostics, name the used features according to the hardware manual. Use a dash ("-") if not applicable.	▸ RAM check of message buffers during startup by software. ▸ Usage of hardware feature "controller register self-test".
Specifics	Document specifics that are not obvious in the hardware manual, such as deviating hardware usage due to hardware issues or special optimizations agreed with the semiconductor vendor or results of clarification of ambiguous parts of the hardware manual. Use a dash ("-") if not applicable.	▸ Module directly accesses the corresponding registers within the Interupt Controller to enable / disable the WAKEUP interrupt.

VECTOR

Hardware Software Interface – Registers

- For register and mask / bit definitions use the names as specified in the hardware manual.
- Document every register definition (e.g. structure member) with a doxygen comment containing the following information:

Property	Description	Example
Description	Short description of the register as found in the hardware manual.	▸ Message Identifier Low Register
Unused functionality	Document unused functionality of the defined register. Omit if all functionality of the register is used.	▸ Bits 5..7 (Timestamping) are not used.
Privileged mode	Describe if the register (or parts of it) may be accessed only in a privileged or otherwise protected mode.	▸ Bits 2..16 (Prescaler): ENDINIT Protection ▸ Write access only in privileged mode. ▸ Access only from trusted zone.
Specifics	Document specifics that are not obvious in the hardware manual.	

- Group register mask / bit definitions by the corresponding register and place a doxygen comment referring to the registers name above this group.
- Document every register mask / bit definition with a doxygen comment containing the following information:

Property	Description	Example
Description	Short description of the mask / bits definition.	▸ Transmit Request ▸ Operation Mode

VECTOR

Hardware Software Interface – Example

```
/*!
 * \internal
 *  Hardware manuals:
 *    R01UH0076ED200 Rev.2.00, FCN
 *    V850E2M User's Manual: Architecture Rev.1.00
 *  Errata sheets: -
 *  Access mechanism: Memory mapped registers
 *  Used registers: See definitions below
 *  Hardware features related to independence or partitioning: -
 *  Operating modes: Refer to the states specified in the CAD
 *  Hardware diagnostics: RAM check of message buffers by software
 *  Specifics:
 *    Access to message buffers in SLEEP mode is not checked at ISR entry.Instead, interrupts are cleared after SLEEP
 *    mode was entered.
 * \endinternal
 */

/*! CTRL register bit masks */
#define CAN_CTRL_TRQ                0x0002   /*!< Transmit Request */
#define CAN_CTRL_DN                 0x0004   /*!< Rx Data New */
#define CAN_CTRL_IE                 0x0008   /*!< Interrupt Enable */

/*! Registers of message buffers corresponding to memory layout */
typedef volatile struct Can_MessageBufferStruct
{                                  /* @8000h + 0x10 * m */
  uint8  DATA[8];      /*!< +00h Message Data Byte 0..7 Register */
  uint8  DLC;          /*!< +08h Message Data Length Code Register */
  uint8  notUsed;      /*!< +09h Access Prohibited */
  uint16 IDL;          /*!< +0Ah Message Identifier Low Register  */
  uint16 IDH;          /*!< +0Ch Message Identifier High Register */
  uint16 CTRL;         /*!< +0Eh Message Control Register.
                                Bits 4..7 are always set to 0 (ABT is unused).
                                Write only in TZ mode. */
} Can_MessageBufferType;
```

VECTOR

Hardware Software Interface – User Documentation

Provide the following information in the Technical Reference:

- Reference to the used hardware manual (incl. version) and the relevant chapter / page / peripheral name.
- Configuration aspects if applicable, such as:
  - Usage of hardware instances
  - Base address
  - Corresponding peripheral names or register sets
- Special hardware aspects if applicable, such as:
  - Considered errata sheets
  - Deviations from hardware manual
  - Usage of “foreign” registers
  - Necessity of privileged mode

Hardware Software Interface – Safety Manual

- If a safety manual is provided for a hardware that the component supports:
  - Implement relevant requirements.
  - Refer to the hardware's safety manual(s) from within the "Hardware Manuals" section of the HSI doxygen comment.
  - Within the component's safety manual:
    - Refer to the hardware's safety manual.
    - Document which requirements from the hardware's safety manual are fulfilled by the component.
- Ask a safety coach for assistance how to implement requirements from safety manuals.

For general information about the hardware software interface also refer to:
https://vglobpessvn1.vg.vector.int/svn/Products/MICROSAR4/ProductFeatures/PF4_SafeBSW/trunk/90_ProjectManagement/20_Projekt_1_Safety_Requirements/MICROSAR_Hardware_Software_Interface_Specification.pdf

VECTOR

Configuration

- Document all configuration parameters within the BSWMD file.
- Document all value ranges of configuration parameters within the BSWMD file.
- Dependencies between configuration parameters are checked by the generator tool validations. The documentation of these dependencies is part of the generator tool documentation.
- The BSWMD file must fulfill “BSWMD Look And Feel”:
  See Z:\PES\PES8\PES8.3_Team\VISEM\BSWMD-Description-Scanner\RegelnBSWMDDescription.docm

VECTOR

Template

- The template follows and contains many design patterns.
- The template includes examples for doxygen design documentation.
- The template considers the formatting rules.
- The template is statically checked by QAC, PCLint and CDK.
- Usage of the template is…
  - …mandatory as basis for new components.
  - …a source of examples if existing components are refactored.

- Available here: MsnSafeBsw.c, MsnSafeBsw.h

VECTOR

Formatting

- Summary of most important guidelines from WI_C-StyleGuide_Std_Comp_Dev_PES.pdf
  - One code indentation level is <space><space>
  - For single and multi-line comments refer to the examples in the code template
  - Column limit is 120
  - For the file and function prolog refer to the examples in the code template
  - Formatting of operators and expressions general rules (details s. GS:028)
    - Unary operators are not separated by <space> from the associated expression
    - Binary operators are separated with <space> from both associated expressions
  - Formatting of integers
    - Hex: 0xAFFEU
    - Dec: 29734, 29734U
  - Pre-processor indentation by <space> per level between '#' and <PreProcStatement>
    - Example: # if defined (MIP_ABC)

Migration procedure

- Implement the rules of this WG…
  - …for additionally created C-functions for new feature development
  - …when restructuring of existing C-functions due to new feature development
- **Do not** implement the rules of this WG if e.g. only a bug is fixed, loop conditions are changed, or a function call is added, etc.

- **Only** perform refactoring of existing, unmodified code to comply with these rules, if **all** of the following criteria are met:
  - 100 % variant coverage, and
  - more than 80 % condition/decision coverage over all configurations. (Do not confuse with test end criteria for SafeBSW.)

- Structural refactoring must lead to a
  - Better describable implementation (design), and
  - Better testable implementation
- **Do not** build sub-structures **only** for metrics improvement.

For more information about Vector
and our products please visit

www.vector.com

Author:
Wolf, Jonas
Vector Germany