<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Changed by</th>
<th>Change Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.10.2010</td>
<td>1.2.0</td>
<td>AUTOSAR Administration</td>
<td>• Removal of Footnotes on p 33 &amp; 37</td>
</tr>
<tr>
<td>30.11.2009</td>
<td>1.1.0</td>
<td>AUTOSAR Administration</td>
<td>• Extended to cover more kinds of tests</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Improved guidelines and examples</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Removed references to non-public documents</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Re-aligned with other AUTOSAR conformance documents</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Legal disclaimer revised</td>
</tr>
<tr>
<td>23.06.2008</td>
<td>1.0.1</td>
<td>AUTOSAR Administration</td>
<td>Legal disclaimer revised</td>
</tr>
<tr>
<td>14.11.2007</td>
<td>1.0.0</td>
<td>AUTOSAR Administration</td>
<td>Initial Release</td>
</tr>
</tbody>
</table>
Disclaimer

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

The material contained in this specification is protected by copyright and other types of Intellectual Property Rights. The commercial exploitation of the material contained in this specification requires a license to such Intellectual Property Rights.

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

The AUTOSAR specifications have been developed for automotive applications only. They have neither been developed, nor tested for non-automotive applications.

The word AUTOSAR and the AUTOSAR logo are registered trademarks.

Advice for users

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

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

1 Document Overview ............................................................................................................ 5
   1.1 Focus and Scope ........................................................................................................ 5
   1.2 How to use the Document ........................................................................................ 5
   1.3 Abbreviations ............................................................................................................ 5
   1.4 References .................................................................................................................. 6
2 The CTSpec Creation and Validation Process ....................................................................... 7
3 Analysis Phase ................................................................................................................... 8
   3.1 Collect Test Input Baseline ........................................................................................ 8
   3.2 Preparation for Analysis ............................................................................................ 8
   3.3 Analyze Specification Items ....................................................................................... 9
   3.4 Categorize Specification Items .................................................................................. 11
      3.4.1 No Requirement .................................................................................................. 14
      3.4.2 Redundant Requirement ................................................................................. 14
      3.4.3 Informal Requirement ..................................................................................... 14
      3.4.4 Definition of Configuration Parameter ............................................................ 14
      3.4.5 Implementation of Configuration Parameter ..................................................... 15
      3.4.6 Requirement on Configuration ......................................................................... 15
      3.4.7 Detection of Wrong Configurations ................................................................. 15
      3.4.8 Development Error ............................................................................................ 16
      3.4.9 Header Files for Internal Use ............................................................................. 16
      3.4.10 Inside Source Code ......................................................................................... 17
      3.4.11 Header Files for External Use ........................................................................... 17
      3.4.12 Provided Signature / Required Signature ......................................................... 17
      3.4.13 Requirement on Module Behavior .................................................................... 18
      3.4.14 Requirement on Re-entrance of Module .......................................................... 18
      3.4.15 Requirement on Execution in Interrupt Context .............................................. 18
      3.4.16 Requirement on Other Module ......................................................................... 18
      3.4.17 Direct Hardware Access .................................................................................... 19
      3.4.18 Non-observable Module Behavior .................................................................... 19
      3.4.19 Non-testable Requirement .............................................................................. 19
      3.4.20 Vendor-Specific Extensions .......................................................................... 20
      3.4.21 Pending on Bug ............................................................................................... 20
   3.5 Associate Test Method with Test Category ................................................................... 21
3.6 Review of SWS Analysis Results .................................................................................... 21
3.7 Delivery of the Refined SWS Document ......................................................................... 22
4 Design Phase ..................................................................................................................... 23
   4.1 Types of Conformance Tests ...................................................................................... 23
      4.1.1 Static Conformance Test Cases ......................................................................... 23
      4.1.2 Dynamic Conformance Test Cases .................................................................... 24
   4.2 Test Case Identification .............................................................................................. 25
      4.2.1 Definition of a unique Test Case Identifier .......................................................... 25
      4.2.2 Definition of the Test Purpose .......................................................................... 25
      4.2.3 Definition of Test Procedures ........................................................................... 26
      4.2.4 Definition of Additional Conditions ................................................................. 26
   4.3 Test Case Design ......................................................................................................... 27
      4.3.1 Test Architecture ............................................................................................... 27
      4.3.2 Function Decomposition .................................................................................... 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.3</td>
<td>Specification of Configuration Sets</td>
<td>33</td>
</tr>
<tr>
<td>4.4</td>
<td>Preparation of Test Case Specification</td>
<td>42</td>
</tr>
<tr>
<td>5</td>
<td>Implementation Phase</td>
<td>43</td>
</tr>
<tr>
<td>5.1</td>
<td>BSW Module Simulation</td>
<td>43</td>
</tr>
<tr>
<td>5.2</td>
<td>Dynamic Test Cases</td>
<td>44</td>
</tr>
<tr>
<td>5.3</td>
<td>File Structure</td>
<td>44</td>
</tr>
<tr>
<td>6</td>
<td>Validation Phase</td>
<td>45</td>
</tr>
<tr>
<td>6.1</td>
<td>Test Case Validation using a Simulation of the BSW Module</td>
<td>45</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Motivation</td>
<td>46</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Validation Setup</td>
<td>46</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Error Tracing during Validation</td>
<td>47</td>
</tr>
<tr>
<td>6.2</td>
<td>The Validation Workshop</td>
<td>47</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Preparation for the Validation Workshop</td>
<td>47</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Conducting the Validation Workshop</td>
<td>47</td>
</tr>
<tr>
<td>6.3</td>
<td>Results of the Validation Workshop</td>
<td>47</td>
</tr>
<tr>
<td>6.4</td>
<td>Validation against Misbehavior</td>
<td>48</td>
</tr>
<tr>
<td>7</td>
<td>Formal Descriptions of Test Cases</td>
<td>50</td>
</tr>
<tr>
<td>7.1</td>
<td>Data Type Mapping</td>
<td>50</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Guideline</td>
<td>50</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Mapping Rules</td>
<td>50</td>
</tr>
<tr>
<td>7.2</td>
<td>Configuration Mechanism</td>
<td>52</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Data types of Test Parameters</td>
<td>53</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Format of Test Parameters</td>
<td>53</td>
</tr>
<tr>
<td>7.3</td>
<td>Control Part</td>
<td>54</td>
</tr>
<tr>
<td>7.4</td>
<td>Handling Open Implementations</td>
<td>54</td>
</tr>
<tr>
<td>7.5</td>
<td>Pointer Handling</td>
<td>54</td>
</tr>
<tr>
<td>7.6</td>
<td>Error Condition Handling</td>
<td>57</td>
</tr>
<tr>
<td>7.7</td>
<td>TTCN-3 Coding Style</td>
<td>57</td>
</tr>
</tbody>
</table>
1 Document Overview

This document describes the methodology to create and validate AUTOSAR BSW Conformance Test Specifications (CTSpecs) in general. It describes how to realize the CTSpec creation process and details potential tool support. Furthermore the document specifies how to apply the TTCN-3 test methodology to testing the conformance of AUTOSAR BSW module implementations. This includes defining the architecture of the CTSpecs and specifying the relevant execution environment components. Hence, the document is structured into 3 parts:

1. Introduction (this chapter)
2. Realization of the CTSpec creation process (Chapters 2 to 6)
3. Details on the formalization of the CTSpecs (Chapter 7)

1.1 Focus and Scope

This document provides a detailed foundation for creating CTSpecs. It focuses more on the details of the work activities and less on the overall creation process. An overview of the CTSpec creation process and the description of the roles involved and their interactions are given in [3].

1.2 How to use the Document

This document presupposes a basic understanding of the CTSpec creation process, the roles involved and the associated terminology. These are all described in the CTSpec Process Overview document ([3]). The CTSpec Execution Constraints document ([4]) is a supplement that discusses issues related to deploying and executing CTSpecs after they have been created.

1.3 Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Program Interface</td>
</tr>
<tr>
<td>BSW</td>
<td>Basic Software</td>
</tr>
<tr>
<td>CC</td>
<td>Conformance Class</td>
</tr>
<tr>
<td>CD</td>
<td>Coder/Decoder (TTCN-3 – see Part 4)</td>
</tr>
<tr>
<td>CH</td>
<td>Component Handling (TTCN-3 – see Part 4)</td>
</tr>
<tr>
<td>CTA</td>
<td>Conformance Test Agency</td>
</tr>
<tr>
<td>CTSpec</td>
<td>Conformance Test Specification</td>
</tr>
<tr>
<td>CTS</td>
<td>Conformance Test Suite</td>
</tr>
<tr>
<td>DEM</td>
<td>Diagnostic Event Manager</td>
</tr>
<tr>
<td>DET</td>
<td>Development Error Tracer</td>
</tr>
</tbody>
</table>
### Abbreviation Description

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU</td>
<td>Electronic Control Unit</td>
</tr>
<tr>
<td>ICC</td>
<td>Implementation Cluster Conformance Class</td>
</tr>
<tr>
<td>ICS</td>
<td>Implementation Conformance Statement</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>PA</td>
<td>Platform Adapter (TTCN-3 – see Part 4)</td>
</tr>
<tr>
<td>PS</td>
<td>Product Supplier</td>
</tr>
<tr>
<td>RM</td>
<td>Requirements Management</td>
</tr>
<tr>
<td>RTE</td>
<td>Run Time Environment</td>
</tr>
<tr>
<td>SA</td>
<td>System Adapter (TTCN-3 – see Part 4)</td>
</tr>
<tr>
<td>SUT</td>
<td>System Under Test</td>
</tr>
<tr>
<td>SW-C</td>
<td>Software Component</td>
</tr>
<tr>
<td>SWS</td>
<td>Software Specification</td>
</tr>
<tr>
<td>TE</td>
<td>TTCN-3 Executable (TTCN-3 – see Part 4)</td>
</tr>
<tr>
<td>TM</td>
<td>Test Manager (TTCN-3 – see Part 4)</td>
</tr>
<tr>
<td>TRI</td>
<td>TTCN-3 Runtime Interface (TTCN-3 – see Part 4)</td>
</tr>
<tr>
<td>TTCN-3</td>
<td>Testing and Test Control Notation, version 3</td>
</tr>
</tbody>
</table>

### 1.4 References

[1] TTCN-3 specifications  
[http://www.ttcn-3.org/Specifications.htm](http://www.ttcn-3.org/Specifications.htm)

AUTOSAR_PD_BSWCTSpecBackground.pdf

AUTOSAR_PD_BSWCTSpecProcessOverview.pdf

AUTOSAR_PD_BSWCTSpecExecutionConstraints.pdf

AUTOSAR_SWS_PlatformTypes.pdf

AUTOSAR_SWS_DevelopmentErrorTracer.pdf

AUTOSAR_TR_CImplementationRules.pdf

2 The CTSpec Creation and Validation Process

Roles perform the activities in the CTSpec creation process. The process can be organized depending on how persons and/or organizations adopt the following roles ([3]):

- Test Designer
- Test Implementer
- Test Validation Implementer
- Test Assessor

This document specifies how the process should be put into practice. It describes the process activities in more detail to enable their realization. From an activity point of view, four work phases can be identified in the CTSpec creation process:

- **Analysis Phase**: Analysis and refinement of the SWS document
- **Design Phase**: Test case identification and design of the CTSpec
- **Implementation Phase**: Implementation of test cases and module simulations
- **Validation Phase**: Validation of test cases

![Figure 1 – Phases and activities of the CTSpec creation process](image)

These phases are discussed in the following chapters.
3 Analysis Phase

The objective of the analysis activities, as described in the following sections, is to produce a refined SWS document that is ready for test case identification. The Test Designer refines the SWS to be conformance tested. This refinement mainly consists of requirements engineering activities on the SWS document. A requirements management (RM) tool is recommended to be used in order to automate change and version management, traceability among and between requirements and test cases and for working concurrently on and exchanging work results. This section assumes the use of an RM tool throughout.

3.1 Collect Test Input Baseline

First, all relevant documents and their proper versions have to be identified. This includes at least the following documents:

- the SWS of the module itself,
- the SWSs of all neighboring modules that contain requirements that must be satisfied by the module,
- any deviation sheet for the module

The baseline must be documented with the names of the documents, their versions and dates. The baseline for the NvM in the CTSpec Creation Pilot was:

<table>
<thead>
<tr>
<th>Specification</th>
<th>Version</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification of NVRAMManager</td>
<td>2.0.0</td>
<td>28.04.2006</td>
</tr>
<tr>
<td>AUTOSAR BSW Deviations Validator 2</td>
<td>107</td>
<td>02.08.2006</td>
</tr>
<tr>
<td>Specification of ECU Configuration Parameters</td>
<td>0.08</td>
<td>11.06.2006</td>
</tr>
<tr>
<td>Specification of Module Memory Abstraction Interface</td>
<td>1.0.0</td>
<td>23.03.2006</td>
</tr>
<tr>
<td>Specification of Module EEPROM Abstraction</td>
<td>1.0.0</td>
<td>23.03.2006</td>
</tr>
<tr>
<td>Specification of Flash EEPROM Emulation</td>
<td>1.0.0</td>
<td>23.03.2006</td>
</tr>
<tr>
<td>Specification of CRC Routines</td>
<td>2.0.0</td>
<td>28.04.2006</td>
</tr>
</tbody>
</table>

3.2 Preparation for Analysis

In this step, all specification items in the baseline documents must be linked or copied into the RM tool\(^1\). At least the following attributes must be assigned to the module requirements:

<table>
<thead>
<tr>
<th>ID given by the RM tool</th>
<th>Original requirements ID from the SWS</th>
</tr>
</thead>
<tbody>
<tr>
<td>For referring to the refined SWS requirement.</td>
<td>For referring to the associated original SWS requirement.</td>
</tr>
</tbody>
</table>

\(^1\) The exact procedure depends on the RM tool being used. For example, DOORS requires copying the requirements into the DOORS repository.
### Related configuration parameter
The configuration parameter related to the refined SWS requirement (if applicable). This attribute is later used in configuration analysis and for creating configuration sets.

### Category
The category assigned to the refined SWS requirement. Test cases are later identified based on these categories (as defined in Chapter 3.4).

### References to test cases that cover the requirement
The ID(s) of the test case(s) that test the requirement. This attribute realizes the traceability between refined SWS requirements and test cases.

### Analysis comment
Additional information or comments that may arise.

While the values of the first and second attributes can be defined during preparation, the values of the remaining attributes are defined in subsequent analysis activities.

The following general guidelines must be followed during this preparation activity:

- Keep the overall order of the SWS requirements in the RM tool the same as the order of the items in the SWS specification.
- All labeled SWS items must appear.
- Ensure that there is an entry for each configuration parameter in the module (typically from chapter 10.2). Document the name of the configuration parameter and add its description. Enter the SWS-Req-ID when it is available.
- Ensure that there is an entry for the signature of each function provided or required by the module.
- Ensure that the general properties of each API-call as defined in the SWS (e.g. reentrancy, synchronous/asynchronous) are entered in the table.

### 3.3 Analyze Specification Items
The actual analysis must be performed by personnel familiar with the BSW module’s functionality and configuration parameters. The following guidelines must be followed during SWS analysis:

- If an SWS item is not atomic\(^2\) (i.e. if comprising several requirements), divide it into atomic items and tag them with the ID of the original specification item extended with letters (a, b …). For example:

  \[
  \begin{align*}
  \text{NVM449a} & \quad \text{void NvM\_GetDataIndex(NvM\_BlockIdType BlockId, uint8* DataIndexPtr)} \\
  \text{NVM449b} & \quad \text{NvM\_GetDataIndex operation is Reentrant.} \\
  \text{NVM449c} & \quad \text{The service NvM\_GetDataIndex is for getting the currently set DataIndex of a dataset NVRAM block.}
  \end{align*}
  \]

- Ensure that an SWS item’s content is understandable and self-describing. If necessary, add context information.

---

\(^2\) In general, an atomic requirement refers to a single property of the BSW module.
- Relate an SWS item with a configuration parameter if it affects this particular SWS item.
- Ensure necessary and sufficient pre- and post-conditions for each function provided or required by the module. If necessary, add entries even if not having explicit SWS item IDs.
- Add other information from the SWS that is relevant to conformance testing, even if not having explicit SWS item IDs.
- Add test-specific IDs to SWS items not identified yet by adding CT and a unique number to module’s short name, for example:

<table>
<thead>
<tr>
<th>Additional Information Type</th>
<th>Action To Be Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information only, for any SWS</td>
<td>No need to capture the paragraph, since no test will result</td>
</tr>
<tr>
<td>Redundant to a requirement in the current SWS</td>
<td>No need to capture the paragraph, since the requirement already exists</td>
</tr>
<tr>
<td>Information needed to understand another requirement</td>
<td>Capture the paragraph as an analysis comment in the other requirement</td>
</tr>
<tr>
<td>Paragraph contains a requirement on the current SWS</td>
<td>To be captured as a separate requirement and given a test-specific ID</td>
</tr>
<tr>
<td>Paragraph contains a requirement on another SWS</td>
<td>No need to capture the paragraph, since it will be captured during analysis of the neighboring modules of the other SWS</td>
</tr>
</tbody>
</table>

Similar statements hold for requirements on configuration parameters as specified in SWS Chapter 10 “Configuration Specification” because they affect the selection and parameterization of conformance tests related to a concrete module variant (configuration).

Information from SWSs of collaborating modules (neighboring SWSs) that is relevant for conformance tests of the module (SUT) must be considered following similar guidelines. Typically, the entries from the neighboring SWSs can be limited to:

- Pre-conditions on functions used by the module under test.
- Post-conditions on functions provided by the module under test.

In general the test-specific SWS resulting from the SWS analysis must contain all technical requirements on the module under test, including those imposed by collaborating modules.

When adding other information (that has no SWS ID) from the SWS that is relevant to conformance testing, the following actions should be taken for the given classes of added information.
During the activities described so far, it must be continuously ensured that the requirements are complete, understandable, logically correct and free from contradictions. Any misunderstandings, errors, inconsistencies and missing information that emerge in the original SWS document must be raised as bug reports for the owner of the original SWS document. Bug reports must also be raised for additional requirements discovered as part of the analysis phase.

The specification items must be modified based on the resolution of the bug reports:

- When an item is changed, make a note under “analysis comment”.
- When an item becomes obsolete, mark the requirements text appropriately (e.g. by crossing it out) and make a note under “analysis comment”. Do not simply delete the item as it will then not be known whether this item has been overlooked or deleted.
- When additional configuration items or specification items are added, make a note under “analysis comment”.

Finally, where a deviation sheet exists, add relevant entries from the deviation sheet according to the following guidelines:

- When the deviation item changes an SWS item, integrate the deviation item in the existing SWS item (i.e. add the deviation number and change the description so that it reflects the contents of the deviation sheet).
- When the deviation is an addition or a more complex modification, add an additional line to the SWS analysis document at a logical place (next to the affected SWS-item or configuration parameter).

The activities up to now have produced a well structured and ordered SWS document that contains all specification information related to the BSW module. Incorporating an RM tool has prepared the refined SWS document well for use in the remaining phases of the CTSpec creation process.

### 3.4 Categorize Specification Items

In general, the refined SWS document contains many specification items that are in fact not relevant for conformance testing. Each specification item must therefore be categorized to establish whether and how it is relevant to conformance testing. The analysts must understand the schema well and apply it accurately.

The following information can be derived directly from a specification item’s category:

- Relevant for conformance testing?
  - Yes / No
- How to test
  - TTCN-3 test case / Inspection / …

The following table describes the specification item categories that AUTOSAR has identified so far, their relevance to conformance testing and the appropriate test approach. More information is given in the sections following the table.
<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
<th>Relevance to CT</th>
<th>How to test</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Requirement</td>
<td>The SI is too vague.</td>
<td>no</td>
<td>irrelevant</td>
</tr>
<tr>
<td>Redundant Requirement</td>
<td>The SI is already covered by one or several other SIs.</td>
<td>no</td>
<td>irrelevant</td>
</tr>
<tr>
<td>Informal Requirement</td>
<td>The SI contains an informal description on the module. This description</td>
<td>no</td>
<td>irrelevant</td>
</tr>
<tr>
<td></td>
<td>usually contains non-functional requirements.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Definition of Configuration Parameter</td>
<td>The SI is a definition of a configuration parameter.</td>
<td>yes</td>
<td>Configuration Inspection</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Implementation of Configuration Parameter</td>
<td>The SI defines how a configuration parameter is realized (configuration</td>
<td>no</td>
<td>irrelevant</td>
</tr>
<tr>
<td></td>
<td>class).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Requirement on Configuration</td>
<td>The SI contains either general definitions on the configuration values or</td>
<td>yes</td>
<td>Configuration Inspection</td>
</tr>
<tr>
<td></td>
<td>direct constraints on permitted configuration values.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Detection of Wrong Configurations</td>
<td>The SI contains demands on how to detect wrong configurations or which</td>
<td>no</td>
<td>irrelevant</td>
</tr>
<tr>
<td></td>
<td>configurations are wrong.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Development Error</td>
<td>The SI defines behavior directly related to the Development Error Tracer</td>
<td>no</td>
<td>Dynamic test case</td>
</tr>
<tr>
<td></td>
<td>(DET).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Header Files for Internal Use</td>
<td>The SI contains definitions of header files that are used by the module</td>
<td>no</td>
<td>irrelevant</td>
</tr>
<tr>
<td></td>
<td>only.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inside Source Code</td>
<td>The SI contains structural definitions of the source code that are not</td>
<td>no</td>
<td>irrelevant</td>
</tr>
<tr>
<td></td>
<td>related to functionality.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inside Header File</td>
<td>The SI contains structural definitions of the header file not related to</td>
<td>no</td>
<td>irrelevant</td>
</tr>
<tr>
<td></td>
<td>functionality.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Header Files for External Use</td>
<td>The SI contains definitions of header files that are to be included by</td>
<td>yes</td>
<td>Source Code Inspection</td>
</tr>
<tr>
<td></td>
<td>other modules.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Provided Signature</td>
<td>The SI is a definition of a function that is provided to other modules as</td>
<td>yes</td>
<td>Compile-build process</td>
</tr>
<tr>
<td></td>
<td>API.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3 Development Error tracing is not relevant to conformance testing, but the SIs must be considered in the test suites. See section 3.4.8 for details.
<table>
<thead>
<tr>
<th>Category</th>
<th>Description (SI = specification item)</th>
<th>Relevance to CT</th>
<th>How to test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required Signature</td>
<td>The SI contains demands on functions provided externally.</td>
<td>yes</td>
<td>Compile-build process</td>
</tr>
<tr>
<td>Requirement on Module Behavior</td>
<td>The SI defines module behavior and functionality.</td>
<td>yes</td>
<td>Dynamic test case</td>
</tr>
<tr>
<td>Requirement on Re-entrance of Module</td>
<td>The SI defines behavior and functionality of the module related to reentrancy.</td>
<td>no</td>
<td>irrelevant</td>
</tr>
<tr>
<td>Requirement on Execution in Interrupt Context</td>
<td>The SI defines behavior and functionality of the module related to execution in interrupt context.</td>
<td>no</td>
<td>irrelevant</td>
</tr>
<tr>
<td>Requirement on other Module</td>
<td>The SI is not a requirement on the “module under specification” but a requirement on another module.</td>
<td>no</td>
<td>irrelevant</td>
</tr>
<tr>
<td>Direct Hardware Access</td>
<td>The SI defines behavior and functionality that involves direct access to hardware devices.</td>
<td>no</td>
<td>irrelevant</td>
</tr>
<tr>
<td>Non-observable Module Behavior</td>
<td>The SI defines behavior that cannot be tested by means of the specified API.</td>
<td>no</td>
<td>irrelevant</td>
</tr>
<tr>
<td>Non-testable Requirement</td>
<td>The SI is a requirement on module behavior but cannot be tested due to current limitations in the test approach.</td>
<td>no</td>
<td>irrelevant</td>
</tr>
<tr>
<td>Vendor-Specific Extensions</td>
<td>The SI is a definition on possible extensions to be done by the vendor.</td>
<td>no</td>
<td>irrelevant</td>
</tr>
<tr>
<td>Pending on Bug</td>
<td>The SI has been identified as unclear, clarification has been requested. After clarification, the proper category will be defined for the SI.</td>
<td>undefined</td>
<td>undefined</td>
</tr>
</tbody>
</table>

Table 1 – Overview of the categories for specification items
The following sections explain the specification item categories in more detail.

3.4.1 **No Requirement**

Some items are not requirements because they are too general and/or too weak.

Example: The following is formulated too weakly (“reasonably short”):

| NVM156 | Depending on implementation, callback routines provided and/or invoked by this module may be called on interrupt level. The module providing those routines therefore has to make sure that their runtime is reasonably short. |

The vagueness of a requirement as given in the example shall be indicated by an issue report.

3.4.2 **Redundant Requirement**

Some items are relevant to conformance testing, but are already covered by other items. The comment in the SWS analysis document must contain precise information about which items cover the redundant item.

3.4.3 **Informal Requirement**

The specification item is either a general definition of terms or conditions (e.g. operating conditions) or it describes basic functionality on a very high level. In both cases, the item cannot be directly related to functional behavior that is testable with test cases because it is not specific enough. The specification item is thus not directly relevant to behavioral conformance but must usually be considered when interpreting other items.

3.4.4 **Definition of Configuration Parameter**

The specification item defines the meaning and/or use of a configuration parameter. The selection of correct values is important for the generation of correct module variants as well as for proper parameterization of conformance test suites (see Chapter 4.3). Therefore, these specification items are relevant to conformance testing. However, the correct use of configuration parameters cannot, in general, be verified by automated test cases since knowledge of their application context and knowledge of the scope of the requirements actually supported by a module implementation are required.

Example: The following must be considered when specifying conformance tests for implementations of the NvM module:

| NVM029 | NVM_DATASET_SELECTION_BITS: Defines the number of least significant bits which shall be used to address a certain dataset of a NVRAM block within the interface to the memory hardware abstraction. |
3.4.5 Implementation of Configuration Parameter

The specification item is related to implementation aspects of module configuration parameters. For example, if SWS Chapter 10 specifies the class of a configuration parameter as pre-compile time then it is probably implemented as a C-preprocessor macro (\#define) that becomes a literal in the text section of a module’s object file.

The implementation of a configuration parameter or, in other words, the time at which a module can be configured (pre-compile time, link time, post-build/run-time), affects the binary interface of a module and hence the module integration and setup process, but not the behavior of a correctly configured module. When an SWS specifies that a module parameter shall be configurable without recompiling the module (i.e. configurable at link time or post-build/run-time) but the configuration parameter is implemented as a pre-compile time parameter, then the parameter’s implementation deviates from the specification. This violates the specification even when the behavior of the test object is not violated. Yet, for correct behavior, it is important to specify correct configuration parameter values during test preparation.

Example: The following specification is not relevant for behavioral conformance:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Pre-compile time</th>
<th>Link time</th>
<th>Post-build time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANNM_CHANNEL_ACTIVE</td>
<td>x</td>
<td>x</td>
<td>--</td>
</tr>
</tbody>
</table>

3.4.6 Requirement on Configuration

The specification item is a requirement for a correct module configuration but not on the behavior of the module. Dynamic conformance tests rely on correct module variants and hence on correct names, types, multiplicity (availability) and possible constraints among various module configuration parameters. Module configuration parameters control the selection and execution of conformance tests.

Example: The following specification is relevant for conformance tests of gateway variants of CanNm implementations. The configuration parameter is constrained by other configuration parameters and used to select gateway specific conformance tests.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANNM_REMOTE_SLEEP_INDICATION_ENABLED</td>
<td>Boolean</td>
<td>True if remote sleep indication is enabled.</td>
</tr>
<tr>
<td></td>
<td>Dependency</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This feature is required for gateway nodes only.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>It must not be defined if CANNM_PASSIVE_MODE_ENABLED is defined.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>This parameter shall be derived from NM_REMOTE_SLEEP_IND_ENABLED.</td>
<td></td>
</tr>
</tbody>
</table>

3.4.7 Detection of Wrong Configurations

Specification items of this category define the validity of configuration parameter values. They contain constraints on allowed configuration parameter values resulting from, for example, dependencies on other configuration parameters.
This type of specification item is not relevant to conformance testing since it is already the responsibility of the specific BSW module’s configuration tool to verify the correctness of configuration parameter values, and the conformance tests assume correct and test-subject-valid configurations are supplied for all tests.

Example: The following is not relevant to conformance testing:

| NVM089 | NvM.c shall check if the correct version of NvM.h is included. This shall be done by a preprocessor check of the version number NVM_SW_MAJOR_VERSION. |

### 3.4.8 Development Error

These specification items define functionality related to the detection of development errors. Only modules in development mode shall report errors to the Development Error Tracer (DET) module. Hence, this functionality is, strictly speaking, not relevant to conformance testing of production modules.

Example: The following is not relevant to conformance testing:

| NVM188 | If the NVM_DEV_ERROR_DETECT switch is enabled, API parameter checking is enabled. … |

However conformance tests trace development errors because this supports the analysis of test results and debugging conformance tests, e.g., during test validation runs. Hence dynamic conformance tests optionally check for specification items related to Development Error Detection.

- During conformance testing of modules in production mode (simulating modules or production implementations), tests run with a test double for DET, but disabled by default ([Module]_DEV_ERROR_DETECT==OFF). Conformance tests flags a report sent to DET as conformance violation (wrong module configuration).
- During analysis and validation phases tests and module can run with development error tracer enabled ([Module]_DEV_ERROR_DETECT==ON).

### 3.4.9 Header Files for Internal Use

These specification items define header files to be provided and their inclusion structure. “Internal header files” are only used within the BSW module and not by other modules. The specification items are therefore not relevant to conformance testing as the header files do not affect the BSW module’s interaction with its environment.

Example: The following is not relevant to behavioral conformance tests:

| NVM077 | NvM_Cfg.h shall include NvM_Types.h.  
NvM_Types.h shall include Std_Types.h.  
NvM.h shall include NvM_Cfg.h. |
3.4.10 Inside Source Code

These items specify requirements on the contents of source code (*.c) or header (*.h) files. The requirements can vary from the inclusion of certain header files to module-internal implementation and code structure details or the application of programming style guides. Adherence to such requirements can only be verified through inspection of the respective source code or header file but not by examining the executable BSW module. Therefore, such specification items are not in scope of testing for behavioral conformance.

Example: The following items are not relevant to behavioral conformance tests:

<table>
<thead>
<tr>
<th>Item</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>PWM075c</td>
<td>Pwm.c shall include Pwm.h, MemMap.h, Det.h and SchM_Pwm.h.</td>
</tr>
<tr>
<td>LIN054c</td>
<td>The global data types and functions that are only used internally by a LIN driver, are declared in Lin.c.</td>
</tr>
<tr>
<td>LIN023</td>
<td>The module Lin_IRQHandler contains the implementation of interrupt frames. The implementation of the interrupt service routine shall be in Lin.c.</td>
</tr>
<tr>
<td>IPDUM073</td>
<td>The code of the IpduM module, as long as it is written in C, shall conform to the HIS subset of the MISRA C Standard.</td>
</tr>
</tbody>
</table>

3.4.11 Header Files for External Use

These specification items define the contents of header files required by other BSW modules or SW components and specify how they will be provided.

Since the header files are used to exchange information and definitions between module generation processes, they influence the interaction of these BSW modules or SW components and are thus relevant to conformance testing. However, the correctness of the header files can only be verified by inspecting them and not by examining the executable BSW module.

Example: The following is relevant to conformance testing:

<table>
<thead>
<tr>
<th>Item</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVM076b</td>
<td>The NVRAM manager shall provide: An API interface NvM.h providing the function prototypes to access the underlying NVRAM functions.</td>
</tr>
</tbody>
</table>

3.4.12 Provided Signature / Required Signature

These specification items define function signatures (i.e. function name, return data type, parameter names and parameter data types) that are provided or required by the BSW module.

The correct implementation of these function signatures is of course crucial for conformance and is already checked while compiling and linking the BSW module for test into the dynamic test environment. These specification items are therefore relevant to conformance testing but are not tested by dynamic conformance test cases.

Example: The following is relevant to conformance testing:

<table>
<thead>
<tr>
<th>Item</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVM044</td>
<td>void NvM_SetDataIndex(NvM_BlockIdType BlockId, uint8 DataIndex)</td>
</tr>
</tbody>
</table>
3.4.13 Requirement on Module Behavior

This is the most important group of specification items for dynamic conformance tests. They define the functionality that interacts with the environment through the API and/or other kinds of interfaces (such as RAM blocks used for data exchange, hardware interfaces).

Specification items of this type are naturally relevant to conformance testing and are well suited to functional (black-box) testing with TTCN-3 conformance test cases.

Note that Section 3.8.2 of the Conformance Test Specification Background document [2] defines the types of bugs that are not relevant to AUTOSAR conformance. If a specification item defines behavior that falls into an excluded bug category, it will not be tested. All other forms of module behavior defined in an SWS shall be tested by a test case.

Example: The following must be tested by a test case:

| NVM185 | NvM_ReadBlock: On successful enqueuing a request, the request result of the corresponding NVRAM block shall be set to NVM_REQ_PENDING. |

3.4.14 Requirement on Re-entrance of Module

These specification items define whether an API function is “re-entrant” or not. The current consensus is that they are not relevant to conformance testing.

Example: The following is not relevant to conformance testing:

| NVM045 | NvM_SetDataIndex must be reentrant. |

3.4.15 Requirement on Execution in Interrupt Context

This kind of specification item defines whether an API function must have properties that allow its execution when called by an interrupt service routine. The current consensus is that they are not relevant to conformance testing.

Example: The following is not relevant to conformance testing:

| NVM060 | NvM_JobEndNotification: This routine might be called on interrupt level, depending on the calling function. |

3.4.16 Requirement on Other Module

Some specification items actually pose requirements on BSW modules that interface with the module under test. These specification items contain, for example, requirements on how certain API functions shall be used by other modules.

This group of specification items is not relevant for the creation of the CTSpec.

Example: The following is not relevant to conformance testing of NvM module:

| NVM077g | Only NvM.h shall be included by the upper layer. |
3.4.17 Direct Hardware Access

SWS items that relate to interaction with hardware components are especially important. Such SWS items can often only be tested with individually designed hardware interfaces and thus require more effort to test than SWS items that relate to pure software functionality. Whether these specification items are relevant to conformance testing must be decided individually.

Example: The following is not relevant to conformance testing:

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEPROM052</td>
<td>In normal EEPROM mode, the EEPROM driver shall access the external EEPROM by usage of SPI channels that are configured for normal access to the SPI EEPROM.</td>
</tr>
</tbody>
</table>

3.4.18 Non-observable Module Behavior

An SWS may define module behavior that is observable only in development mode, that is, when the DET is in place. For modules in production mode it is assumed that module clients use the module correctly. Agreed interface contracts enable close and efficient cooperation between modules and module clients. On the other side testability is restricted leading to requirements which are not testable in production mode.

Example: An implementation of CanNm in production mode cannot be tested for CANNM140.

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANNM140</td>
<td>After reset the Network Management state shall be set to NM_STATE_UNINIT.</td>
</tr>
</tbody>
</table>

However this is not immediately obvious because a client module can check a CanNm implementation by means of operation CanNm_GetState which returns the state of the module in an output parameter of type Nm_StateType, an enumeration including among others NM_STATE_UNINIT. But according to CANNM191 a CanNm production implementation cannot be checked for being uninitialized. A call of CanNm_GetState would just return a general NM_E_NOT_OK signaling an execution error.

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CANNM191</td>
<td>If not initialized, the NM shall reject every API service apart from CanNm_Init; the called function shall not be executed, but instead of that it shall report NM_E_NO_INIT to the Development Error Tracer and it shall return NM_E_NOT_OK to the calling function.</td>
</tr>
</tbody>
</table>

3.4.19 Non-testable Requirement

Some items are not testable due to (current) limitations in the test equipment, test strategies or test techniques employed. Given test equipment, strategies or techniques of a different nature, it may be that these items could be tested. This category exists to capture those items that are testable in principle, but practically no such test can be written under given constraints.
Example:

| ICU090 | The function Icu_SetActivationCondition shall set the activation-edge according to Activation parameter for the given channel. |

The analysis comment relating to an SWS item that is categorised as Non-testable Requirement contains a brief description of a test that could be performed given other test equipment, strategies or techniques, together with the reason why that test cannot be performed. The following would be a comment in the SWS analysis document for the above example:

It would be possible to test this requirement using the following two-step test:

1. Initialize the hardware, initialize the module, set the activation edge, provide a single edge of the non-activating type, and observe no notification.
2. Repeat step 1 with a single edge of the activating type, observe the correct notification.

However, with the current test equipment and techniques it is not practical to provide a single edge as a hardware input to the SUT, synchronized with the test suite.

3.4.20 Vendor-Specific Extensions

The SWS documents leave some room for vendor-specific extensions and definitions such as error codes. This category consists of SWS items that define how this room is to be used. Since the CTSpec can only test standardized functionality, vendor-specific extensions are naturally out of scope of conformance tests.

Example: The following is not relevant to conformance testing:

| NVM186 | Values for production code Event Ids are assigned externally by the configuration of the Dem. They are published in the file Dem_IntErrId.h and included via Dem.h. |

3.4.21 Pending on Bug

The category of these SWS items cannot be determined until a bug entry in AUTOSAR’s Bugzilla system has been clarified. They are therefore categorized temporarily as “Pending on Bug”. After the open issue has been clarified, the category must be changed according to the result.

In general, there are three reasons for raising a bug on an SWS item, and these should be documented in the analysis as listed below.

For SWS items where the item itself is unclear, a comment should state:

- Analysis is not possible due to problems with the SI. See bugs referenced in the "Bugzilla Reference" column.

For SWS items where the item is clear, but issues with other items mean this one cannot be properly categorized, a comment should state:

- Categorization is not possible due to problems with related SIs. See bugs referenced in the "Bugzilla Reference" column.
For SWS items where the item is clear and can be properly categorized, but issues with other items mean it is not yet possible to implement a test for this item, a comment should state:

- Implementation is not possible due to problems with related SIs (see bugs referenced in the "Bugzilla Reference" column). Categorization will be <category> when the bugs are resolved.

In this last case the comment could also include notes on the implementation, if it is non-trivial or non-obvious, to aid with subsequent process steps and potentially with maintenance.

### 3.5 Associate Test Method with Test Category

The “How to test” column in Table 2 below indicates how to test specification items for conformance. The second column details the different ways of ensuring conformance:

<table>
<thead>
<tr>
<th>How to test</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuration Inspection</td>
<td>The specification item has to be considered when the configuration sets for CTSpec execution are being defined and/or generated. In particular, the meaning of and interdependencies among configuration parameters have to be taken into account.</td>
</tr>
<tr>
<td>Source Code Inspection</td>
<td>Adherence to the specification item is verified by checking for the presence of the specified contents in the affected header file (as part of the SUT). This can be done manually or can be automated.</td>
</tr>
<tr>
<td>Compile-build process</td>
<td>Adherence to the specification item is verified by mechanisms in the (already automated) compile-build process. Typically, violations result in errors which are reported by the tools used (e.g. compile error if API function is defined incorrectly).</td>
</tr>
<tr>
<td>Dynamic test case</td>
<td>A test case implemented in TTCN-3 can be defined to verify adherence to the specification item.</td>
</tr>
</tbody>
</table>

Table 2 – Conformance Test Methods

### 3.6 Review of SWS Analysis Results

A person other than the person who did the analysis must review the analysis result. The review criteria can be derived directly from the analysis guidelines given in the previous sections. The refined SWS document produced in the analysis phase must state the names of both the analyst and the reviewer.
3.7 Delivery of the Refined SWS Document

The refined SWS document package must contain the modified specification text, the analysis results/comments and the categorization of all specification items.
4 Design Phase

The design phase takes the output of the analysis phase, the corrected SWS document and the categorized specification items and develops conformance tests: either dynamic test cases that execute the SUT’s code or static test cases that do not execute the SUT’s code.

The Design Phase consists of the following activities:

1. Identification of test cases
2. Design of the test cases

The objectives of the test case identification activity are:

- Identify the test cases based on the testable specification items
- Define the test steps based on the identified test cases
- Specify the rules for generating configuration sets

The objectives of the test case design activity are:

- Specify the architecture for the test cases (test components, ports etc.)
- Specify the behavior of test components that simulate neighboring modules
- Provide definitions (test functions, function signatures, data types etc.) for implementing the test cases

These two activities – although described separately in this chapter – cannot usually be performed sequentially. As the sub-activities are interdependent, (e.g., the test cases influence the test case architecture, test case architecture influences the test steps) these two activities are generally performed in parallel.

4.1 Types of Conformance Tests

4.1.1 Static Conformance Test Cases

Static conformance test cases are not applied to an executing BSW module, but to other parts of the BSW module, for example to header files or module configuration data. Table 3 lists some methods applicable to static conformance tests.

<table>
<thead>
<tr>
<th>Test Item</th>
<th>Test Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header files</td>
<td>Check whether header file is provided / applied.</td>
</tr>
<tr>
<td>Operation signatures in</td>
<td>Parse operation signatures.</td>
</tr>
<tr>
<td>module interface</td>
<td>Link binaries.</td>
</tr>
<tr>
<td>Configuration data</td>
<td>Check conformity to parameter types and constraints.</td>
</tr>
</tbody>
</table>

Table 3 – Generic test methods for non-TTCN-3 test cases
A static conformance test case can be created in one step directly from the SWS item as depicted in Figure 2.

![Figure 2 – Overview of the creation process for "non-TTCN-3" test cases](image)

### 4.1.2 Dynamic Conformance Test Cases

Dynamic conformance test cases check the behavior of BSW modules. The majority of functional requirements relevant to conformance relate to this behavior. These test cases execute against a fully configured BSW module executable and use the API specified in the BSW module’s SWS.

![Figure 3 – Roles, deliverables and tools in the design, implementation and validation phases](image)
Dynamic conformance tests are created in three phases: design, implementation and validation. Figure 3 shows an overview of the different roles involved, their activities, the tools and the work artifacts. These phases produce TTCN-3 test cases together with the sets of configuration and test parameters in TTCN-3 format.

The following chapters describe in detail the steps to be carried out within the three phases.

4.2 Test Case Identification

The test cases and test steps must be defined based on the refined and atomized specification items. This involves four steps:

1. Definition of a unique test case identifier
2. Definition of the test purposes (i.e. test case objectives)
3. Definition of the test procedure (i.e. test cases’ behavior)
4. Definition of additional conditions for the test case (i.e. test cases’ constraints)

4.2.1 Definition of a unique Test Case Identifier

Conformance test cases must be unambiguously identifiable. A unique test case number must therefore be defined whenever a new test case is created. This number shall be preceded by TC_[Module] to associate the test case with a specific BSW module. The test case numbers are not necessarily in sequence as there is always the possibility that certain test cases will be removed from the CTSpec.

Example:

TC_NVM_0020: Test case 20 for testing NvM implementations.

4.2.2 Definition of the Test Purpose

The test purpose contains the well-defined objective of the test cases. It is a short description of what the test case verifies or ensures. Note that the test purpose is not identical to the specification item as the specification item describes what the BSW module does and the test purpose describes what the test case does.

Test purposes often use phrases as for example

Verify / Ensure / Check that ...
... on reception / in the event / under the condition of ...
... does / does not ...
Example:

<table>
<thead>
<tr>
<th>SWS Service</th>
<th>Test Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dio_WriteChannel</td>
<td>TC_DIO_00XX</td>
</tr>
</tbody>
</table>

void Dio_WriteChannel(
    /* IN */ Dio_ChannelType ChannelId,
    /* IN */ Dio_LevelType Level
)

Dio_WriteChannel provides the service to set a level of a channel.

Check that Dio_WriteChannel sets the specified output pin (ChannelId) to the given bit value (Level: STD_HIGH, STD_LOW).

The test purposes must be defined in a simple manner, so as to permit easy separation and handling of the test cases. Additional information must be provided as test background, if the relation between the specification item, test purpose and the test procedure is not obvious.

Example:

Test Background | TC_DIO_00XX |
----------------|-------------|

The output pin (ChannelId) is connected to an input pin (ChannelId2) before the test run. The bit value (Level) written to the output pin must be the level read back at the connected input pin (Level2).

4.2.3 Definition of Test Procedures

Although it is possible to define dynamic test cases directly in the executable test specification language (TTCN-3), the work is divided into test procedure definition and test implementation. This allows the Test Designer to define the basic idea and steps of a test case in form of test procedures which abstract from, for example, syntactical details and handling default test behavior.

Test procedure descriptions are used as a means of communication between the test case design and the test case implementation. This is necessary when test cases are designed and implemented in different project phases, by different persons or organizations (see Chapter 2).

The logic of the test procedures must be defined precisely enough to be capable of being implemented in TTCN-3 and traced back and forth. This is accomplished with a semi-formal notation. The notation used for defining test procedures, its expressions and statements, is defined depending on the usage context and includes keywords and logical expressions known from programming languages like C enriched with test-specific constructs like verification statements and verdict assignments. Refer to Chapter 3 in the CTSpec Background document ([2]) for guidance on the nature and extent of AUTOSAR conformance testing.

4.2.4 Definition of Additional Conditions

Conditions that are relevant to test case execution must be added to the test case bodies to complete the definitions, for example pre-conditions. A pre-condition is a condition that must be fulfilled before the test case is executed, e.g. a module is not initialized when testing an init function like NvM_Init.
4.3 Test Case Design

Test procedures must follow the basic design of the test cases. The test case design specifies the test components, their interfaces and the functional decomposition of the test functions:

<table>
<thead>
<tr>
<th>Test architecture</th>
<th>Definition of test components that stub out functionality of collaborating modules and their interfaces with the module under test.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function decomposition</td>
<td>Decomposition of test functionality following the principle of separation of concern.</td>
</tr>
<tr>
<td>Specification of Configuration Sets</td>
<td>Definition of rules for selecting configuration parameter sets suitable for conformance testing.</td>
</tr>
</tbody>
</table>

4.3.1 Test Architecture

The test architecture defines test components, communication ports and the function signatures for communicating internally and with the system under test.

4.3.1.1 Test Components

Within TTCN-3 test cases, test components execute the test steps. Since several test components can be created within a test case, test steps can be executed in parallel. BSW conformance testing uses test components, e.g. for error reception through the DET and DEM interfaces.

Defining roles for test components makes the test cases more understandable and easier to re-use. There are three roles for test components:

- Test Case Controller (TCCO)
- Test Case Client (TCCL)
- Test Case Stub (TCST)

The following sections define these test component roles.

Component Role: Test Case Controller (TCCO)

The purpose of the TCCO is to control the execution of the test case. The TCCO creates further test components (i.e. the TCCL and any optional TCSTs) and stops them to end the test case gracefully. After the test case completes, the TCCO collects the (local) test verdicts from the other test components and summarizes them in the final test verdict⁴.

The TCCO is the entry point to TTCN-3 test cases. Thus, it is implemented on the MTC (Main Test Component, see [1]). Note that the TCCO does not communicate with the SUT directly.

⁴ The rule for summarizing the test verdicts is defined in the TTCN-3 standard. Basically, the final test verdict can only be “pass” when all (local) test verdicts indicate “pass” as well.
Component Role: Test Case Client (TCCL)

TCCLs actually implement a test case’s purpose. That is, they send stimuli to the SUT and wait for responses from the SUT. They then validate the responses and finally calculate the (local) test verdict (pass, fail or inconclusive). Thus only TCCLs execute test steps involving interaction with the SUT.

TCCLs communicate with the SUT through defined SUT Adapter ports using procedure-based communication methods. They are implemented on a PTC (Parallel Test Component, see [1]) and are created and controlled by the TCCO.

Component Role: Test Case Stub (TCST)

A TCST is needed if the SUT requires some external functionality. The TCST provides this functionality to the degree necessary to run the test case successfully. Thus, the TCST is in general a limited simulation of an external BSW module.

Typically, the TCST provides services to the SUT by waiting for incoming SUT requests and providing appropriate responses. The TCST makes its own (local) test verdict based on the interactions with the SUT and the test purpose.

Test functions implemented by a TCST are, for example:

- Error handling according to the Development Error Tracer (DET)
- Event handling according to the Diagnostic Event Manager (DEM)
- Library functions used by the SUT like CRC calculation

The TCCL is implemented on a PTC (Parallel Test Component, see [1]) and created and controlled by the TCCO, as required by the test purpose.

4.3.1.2 Ports

Ports must be defined for each test component so that the test component can interact with other test components or with the SUT. In TTCN-3, ports can be defined as message, procedure or mixed ports (i.e. message and procedure at the same time). In general, AUTOSAR BSW module conformance testing only uses procedure ports. Details and examples of defining ports can be found in [1].

4.3.1.3 Interfaces

Procedure ports convey test components’ function calls and their parameters. The group of function calls that can be transmitted over a port is referred to as an interface or Application Programming Interface (API) in the case of a standardized interface.

The signatures of function calls that can be transmitted over ports must be defined resulting in interface definitions for conformance test cases. These signatures are either translated directly from the API definitions in the AUTOSAR SWS documents, or, in case of function calls among test components or towards the target adapter (see also [4] for target adapter functions), are specified in the CTSpec itself.
4.3.1.4 Modeling with UML

The TTCN-3 test case design elements described in the previous sections can be represented with UML component diagrams. The CTSpec design should provide at least the following views:

- Test Architecture View
- Test Case Client View
- Test Parameters View

Test Architecture View

This view (Figure 4) shall contain all possible test component interactions with the BSW module under test (SUT) used in the test cases. It shall depict all test component and SUT ports and their interconnections. The interface definitions associated with these ports can be omitted. The structural information contained in this view is helpful when implementing the framework of the CTSpec design in TTCN-3.

Figure 4 – Example: Test Architecture for the NVRAM Manager (NvM)

Test Case Client View

This view (Figure 5) specifies all ports and their interface definitions from the Test Case Client point of view (i.e. the test component that executes the main test steps). This view gives an overview on all possible interaction methods available for the main test steps of a test case. This is helpful when implementing the test steps.
Figure 5 – Example: Test Case Client View of the NVRAM Manager (NvM)

Test Parameters View

This view (Figure 6) presents all AUTOSAR configuration and test specific parameters relevant to the CTSpec. Module-specific and entity-specific parameters (see Chapter 4.3.3) shall be defined separately as attributes of “parameter classes” which have appropriate TTCN-3 data type names.
4.3.1.5 Test System Dynamics

Sequence diagrams reveal the interaction among various test components during a run of a test case. The following sequences describe behavior of test cases based on example test logs.

Scenario: Define the calling mode of a BSW module’s main function

Test cases may define whether calls to the BSW module’s main function are controlled by the test case (see Figure 7) or automatically by the SUT adapter.

Figure 7 – Specifying the calling mode for the BSW module’s main function
Scenario: Invocation of API Functions

Test cases interact with the BSW module under test by invoking its API functions. In Figure 8, the API function `NvM_GetErrorStatus()` is called and returns an error status code in a memory block that has been allocated earlier. After the API function returns, the error status code is read out of the memory block.

![Call to NvM_GetErrorStatus](image)

**Figure 8 – API function invocation and retrieval of the result from the target memory block**

API functions can, in general, be invoked in two directions:

- Exported functions from test case to the BSW module
- Imported functions from BSW module to test case

In Figure 9, the test case first invokes the main function which results in a number of calls to the lower layer API (here: `MemIf_GetStatus()`) and to the CRC module (here: `Crc_CalculateCRC32()`). These calls are received by the TTCN-3 test components (test doubles) in charge of these API functions. The main function returns after these activities have finished.

![Invocation of the main function with subsequent BSW module activities](image)

**Figure 9 – Invocation of the main function with subsequent BSW module activities**
From the test case viewpoint callbacks of the BSW module are equivalent to imported functions. Errors a BSW module reports to DEM or DET are calls of special import functions like `Dem_ReportStatus()`.

### 4.3.2 Function Decomposition

BSW modules with complex functionality usually require conformance test cases with complex, recurring test steps. To reduce redundancy, recurring test steps shall be defined as test functions (also called “base functions” on TTCN-3 level, see Chapter 5.3) that can be called by the test cases.

In order to make the concept of defining these test functions comprehensible, the CTSpec design for complex BSW modules must include either

- a description of the approach to defining the test functions or
- definitions of typical test functions from which the approach can be deduced.

This information enables the Test Implementer to correctly implement the TTCN-3 base functions based on the test functions, to correctly use the base functions in the test cases and to maintain consistency among the base functions.

### 4.3.3 Specification of Configuration Sets

AUTOSAR BSW modules can be configured by parameters to a very high degree. Configuration parameters define various attributes of a BSW module, like job queue sizes, available operations, call back functions, and identifiers. Defining suitable configuration sets is therefore as fundamental to conformance testing as defining the right test objectives and test cases. They are used to generate BSW modules which are configured correctly for test execution. The module can then be tested by a CTSpec that has also been configured with the same configuration set.

The CTSpec cannot define all AUTOSAR configuration parameters generically in advance, however. Some configuration parameters are constrained by the ICS specified by the vendor of the BSW module. Therefore, the configuration sets used for CTSpec execution must be generated individually for each ICS.

In addition to the configuration parameters defined by AUTOSAR, the CTSpecs use test-specific parameters to define different test execution behavior variants. This concept is an efficient way to test different behavioral conditions without increasing the number of test cases.

AUTOSAR configuration parameters and test-specific parameters are defined in two steps:

1. During CTSpec creation, the rules for the configuration generation are defined with as many concrete values as useful and possible for the purpose and limits of the CTSpec creation and validation process.
2. After provision of the ICS, the rules are refined so that the final configuration sets can be generated.
The following sections describe the strategy and process for generating (in two steps) a number of configuration sets which result in good configuration coverage while remaining reasonable for the purpose of creation and validation of conformance tests.

Good configuration coverage ensures that important and critical working conditions of the BSW module are tested by the CTSpec. This ultimately leads to good test coverage of the module’s functionality.

4.3.3.1 Input to Configuration Generation Process

Most standardized configuration parameters are relevant to conformance testing and one or more values must be defined to sufficiently configure the (configurable) module under test. Standardized configuration parameters can be categorized as module-specific configuration parameters (one value per BSW module) or entity-specific configuration parameters when a module handles several individually configured entities, e.g., memory blocks of different types by a NvM module and pins and ports handled by a Dio module. Vendor-specific configuration parameters can naturally not be considered during the creation and validation phase.

As already mentioned above, additionally CTSpecs may define their own test-specific parameters to specify test variants within the test suites. These test-specific parameters are defined in the design phase of the CTSpec creation process [3] and are used to select and parameterize conformance tests.

4.3.3.2 Data Selection Strategy

For most BSW modules, the number of relevant configuration parameters is high enough to make manually defining appropriate configuration sets highly tedious, error-prone and thus practically infeasible. Hence, in general automation and tool support is required to efficiently identify and combine the relevant configuration parameter values and to generate the resulting configuration sets in TTCN-3 and AUTOSAR XML.

Defining combinations of test input data is an inherent problem in the domain of functional testing which has been widely addressed by researchers. There are several appropriate methods (e.g. classification tree method) and some test data generation tools (e.g. CTE, PICT). A best-practice method which AUTOSAR deems adequate for conformance testing BSW modules is defining equivalence classes and combining the resulting configuration values in pair-wise combination resulting in an effective trade-off between configuration coverage and the total number of configuration sets.

The problem of selecting proper values and minimal combinations of module configuration data and test data for a safe SUT coverage is similar for the CTSpec creation and validation phase and for the CTSpec application (execution) phase. However, for validating conformance test specifications configurable BSW module simulators are used. BSW module simulators imitate either correct behavior or incorrect behavior (see also Chapter 6.4). Variants of BSW module simulators (extreme, typical, critical) eligible for validating specified conformance tests are selected on purpose while adhering to the constraints defined for and among module configuration parameters. Configuration coverage shall be close to the coverage achievable with pair-wise test-
ing techniques. Regression tests for revalidation of test suites are executed with the same module variants and unchanged test data.

4.3.3.3 Defining Configurations

Figure 10 presents an overview of the configuration set generation process which consists of four steps with several activities. The steps performed during CTSpec creation are:

- Analysis
- Clustering and Formalization
- Value Selection and Refinement (partially)

The remaining activities of the “Value Selection and Refinement” step are performed when the ICS is provisioned for a module at the beginning of the conformance testing step (test application). Finally configuration value sets are combined and concrete test data are generated as parts of the “Combination and Generation” step.

Figure 10 – Overview of the generation process for configuration sets
The following sections describe the steps and activities required to create configuration sets using the approach outlined in the previous sections.

**Step 1: Analysis**

The analysis step is part of the first two phases (“Analysis Phase” and “Design Phase”) of the CTSpec creation process described in [3]. In this step, the AUTOSAR configuration containers and parameters required by the BSW module are analyzed from the conformance testing point of view.

Configuration containers structure the set of configuration parameters. The multiplicity of a container determines the existence and multiplicity of contained configuration parameters and containers. Additionally, the multiplicity of configuration containers can affect the applicability of test cases (e.g. no PWM channel can be tested if none is configured). The selection of the multiplicity of a configuration container can be treated similarly to the selection of a value for a regular configuration parameter. Therefore, in the following it is not discriminated between a container multiplicity and a parameter.

It has to be determined whether a configuration parameter value can and should be defined during the CTSpec creation process or whether the value depends on execution, hardware or other external conditions. In the latter case, the value can only be determined after the external conditions (e.g. hardware devices for test execution) have been defined, and is therefore out of the scope of the CTSpec creation process.

Test specific test parameters that are identified during test case design are defined and combined with the AUTOSAR configuration parameters. This results in a collection of test parameters. Requirements and constraints on the test parameters can then be identified either directly from the SWS or from insights gained during analysis of the SWS. Special requirements and constraints must be documented.

The remaining steps treat both the AUTOSAR configuration parameters and test specific parameters in the same way and the same automatable combination algorithms can be applied to both types of parameters.

**Step 2: Clustering and Formalization**

In this step, interdependent parameters are identified and are put into clusters. Clustered test parameters are regarded thereafter as a whole.

Some parameters are inherently interdependent and cannot be treated separately when determining test combinations. This happens, for example, when a set of parameters all relate to a specific functionality (e.g. job queuing and prioritization).

In addition to the clustering activity, the requirements and constraints on the test parameters are formalized into expressions that can be processed automatically.

---

In this document, the terminology “AUTOSAR configuration parameters” refers to those parameters for which values can and should be defined during the CTSpec creation process.
Step 3: Value Selection and Refinement

The value selection and refinement step determines the values for both non-clustered and clustered test parameters and refines the requirements and constraints on the test parameters. As depicted in Figure 10, this step is subdivided into activities performed during CTSpec creation and those that must be done after provision of the ICS, i.e., when the CTSpec is to be executed.

Creating equivalence classes can be helpful in finding concrete, representative values for the test parameters. In general, guidelines like the following shall be applied.

| Boolean or enumeration parameters | Select all values except those which are irrelevant a-priori (e.g. not in focus of conformance testing) or otherwise irrelevant to conformance testing. Document these exclusions. |
| Integer parameters               | Define equivalence classes and pick reasonable values from these classes like the lowest value, the highest value and an intermediate value. |

The value combinations of clustered test parameters are defined manually according to the inherent constraints. Each combination is then regarded as a “cluster value”.

After the ICS has been provisioned and checked for validity, previously generated configuration parameter values must be checked or regenerated according to the constraints stated in the ICS. Further ICS requirements and constraints on the test parameters must be added as formal expressions.

Step 4: Combination and Generation

After the ICS has been considered, the configuration sets are generated from the relevant refined test parameters definitions, clusters, selected values and formalized constraints. Using the selected values, pair-wise combinations of test parameters and clusters are generated in the form of TTCN-3 module parameters (see Chapter 7.1). Based on these combinations, the necessary number of containers is generated according to the selected container multiplicities.

4.3.3.4 Example of Configuration Set Creation

This section presents a possible realization approach to generating configuration sets based on the steps described in the previous sections. It uses a table (a spreadsheet, also referred to as combination table) and a custom tool, called Combiner.

The combination table is filled out during the “Analysis”, “Clustering and Formalization” and “Value Selection and Refinement” steps. In the last step, “Combination and Generation”, the custom tool takes the combination table, evaluates the constraints, applies the pair-wise combination algorithm and writes out the generated configuration sets.

Analysis, Clustering and Value Selection and Refinement

Figure 11 shows a combination table with module-specific configuration parameters and their selected values for an NVRAM Manager module.
The first column contains the test parameters (or clusters). The identifiers of AUTOSAR configuration parameters consist of capital letters and underscores (e.g. \texttt{NVM\_JOB\_PRIORITIZATION}). The identifiers of test-specific configuration parameters are denoted by a “CamelCase” notation and are prefixed with “TC” (e.g. \texttt{TCConfigIdBehavior}).

The “Selected Values” columns contain the test parameter (or cluster) values. Clustered test parameters and clustered values are enclosed by curly brackets (“{”, “}”).

The second column in the combination table of Figure 11 provides for the formalized constraints. No interdependencies or constraints have been identified for the module-specific parameters in this example. This topic is discussed in the following sections.

### 4.3.3.5 Constraints among Parameters

One way of dealing with interdependent test parameters is clustering. The cluster parameter values must be selected manually while respecting the given constraints and interdependencies. Some BSW modules have an additional level of interdependence between test parameters that cannot be resolved by clustering, however.

#### Module-specific Parameters and Entity-specific Parameters

In general, two groups of BSW module test parameters can be distinguished:

<table>
<thead>
<tr>
<th><strong>Module Parameters</strong></th>
<th>These are defined once for a BSW module.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Entity Parameters</strong></td>
<td>Some BSW modules have a number of entities (e.g. memory blocks, pins) which must be configured individually and a specific set of configuration parameters must be provided for each of these entities.</td>
</tr>
</tbody>
</table>

The generation of pair-wise combinations for entity-specific test parameters results in a set of differently configured entities. These entities are collected into the same configuration set. In this way, a test case can then iterate through the entities and apply the tests on those entities that are appropriately configured.
Parameter Interdependence

The NVRAM Manager (NvM) has both module-specific and block-specific test parameters. The module-specific parameters are unique per NvM module. There are block-specific parameters for each NvM block controlled by the NvM module.

Example of a module-specific parameter:

\[ \text{NVM\_DATASET\_SELECTION\_BITS} \]: Defines the number of least significant bits used to address a certain dataset of an NVRAM block within the memory hardware abstraction interface.

Example of an entity-specific parameter:

\[ \text{NVM\_NV\_BLOCK\_NUM} \]: Defines the number of NVRAM blocks in a contiguous area according to the block management type.

The number of bits used to address the dataset NVRAM blocks is defined by \[ \text{NVM\_DATASET\_SELECTION\_BITS} \]. However, \[ \text{NVM\_NV\_BLOCK\_NUM} \] is applied to dataset NVRAM blocks and specifies how many NVRAM blocks are available in total.

Obviously, there is interdependence between \[ \text{NVM\_DATASET\_SELECTION\_BITS} \] and \[ \text{NVM\_NV\_BLOCK\_NUM} \]:

- When, for example, \[ \text{NVM\_DATASET\_SELECTION\_BITS} \] is set to 4, only \(2^4 = 16\) dataset NVRAM blocks can be addressed.
- Thus, in this case, it would not be correct to define \[ \text{NVM\_NV\_BLOCK\_NUM} \] to be greater than 16.

Resolving Parameter Interdependence

The logic behind the interdependence between module-specific and entity-specific parameters (e.g. for memory blocks) is extensive. A generic solution to handle this interdependence in configuration set generation is thus required.

It is assumed that certain combinations of module-specific parameters imply certain groups of entities. When defining and combining these entity groups' configuration parameters, dependencies on module-specific parameters must be respected.

A generic solution based on algebra and evaluation of the interdependence between configuration parameters has been developed for creating the configuration sets:

- The constraints that describe the interdependence between module specific and entity specific test parameters are formulated as algebraic expressions.
- When creating the pair-wise combinations for the entity-specific test parameters, only those entity specific parameter values are used that satisfy the given constraints.

Example:

Values selected for pair-wise combination:

\[ \text{NVM\_DATASET\_SELECTION\_BITS} = \{ 2, 4, 8 \} \]
\[ \text{NVM\_NV\_BLOCK\_NUM} = \{ 3, 6, 10, 32 \} \]

The constraint between these two configuration parameters is formulated as:
\[
\text{NVM\_NV\_BLOCK\_NUM} \leq 2^{\text{NVM\_DATASET\_SELECTION\_BITS}}
\]

Applying this constraint results in the following:

- When 2 is selected for `NVM\_DATASET\_SELECTION\_BITS`, only 3 is used for `NVM\_NV\_BLOCK\_NUM` in pair-wise combination.
- When 4 is selected for `NVM\_DATASET\_SELECTION\_BITS`, then the values \{3, 6, 10\} are used for `NVM\_NV\_BLOCK\_NUM` in pair-wise combination.
- When 8 is selected for `NVM\_DATASET\_SELECTION\_BITS`, then all values \{3, 6, 10, 32, 50\} are used for `NVM\_NV\_BLOCK\_NUM` in pair-wise combination.

Exclusions are another form of parameter restrictions. Example for mutually exclusive parameters:

\[
\text{NVM\_BLOCK\_USE\_CRC} == \text{FALSE} \quad \text{and} \quad \text{NVM\_CALC\_RAM\_BLOCK\_CRC} == \text{TRUE}^{6}
\]

The exclusion relations among certain combinations of test parameters values must be defined and taken into account when generating the pair-wise combinations. The exclusion relations must be expressed as pairs using arbitrary combinations of equalities and inequalities.

Example:

\[
\begin{align*}
\text{NVM\_BLOCK\_CRC\_TYPE} & == \text{NVM\_CRC16} \quad \text{and} \quad \text{NVM\_NV\_BLOCK\_LENGTH} \leq 3 \\
\text{NVM\_BLOCK\_CRC\_TYPE} & == \text{NVM\_CRC32} \quad \text{and} \quad \text{NVM\_NV\_BLOCK\_LENGTH} \leq 5
\end{align*}
\]

**Combination Table**

Figure 12 shows the block-specific configuration parameters part of the combination table from Figure 11. The second column now contains the formalized constraints between module-specific and block-specific parameters. The exclusion constraints have been added in rows.

**4.3.3.6 Output of the Configuration Generation Process**

The configuration generation process produces a number of configuration sets in both TTCN-3 and AUTOSAR XML format (for the BSW module simulators). The TTCN-3 format for configuration data is described in Chapter 7.2.2.

---

\(^6\) When CRC is disabled generally, it cannot be enabled for a specific block.
### [BLOCK_PARAMETERS]

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Selected Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NVM BLOCK USE_CRC</strong></td>
<td>TRUE</td>
</tr>
<tr>
<td><strong>NVM CALC RAM BLOCK_CRC</strong></td>
<td>TRUE</td>
</tr>
<tr>
<td><strong>NVM BLOCK_CRC_TYPE</strong></td>
<td>NVM_CRC16</td>
</tr>
<tr>
<td><strong>NVM BLOCK WRITE PROT</strong></td>
<td>TRUE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>NVM BLOCK MANAGEMENT_TYPE</strong>, <strong>NVM_NV BLOCK_NUM</strong>, <strong>NVM_ROM BLOCK NUM</strong></th>
<th><strong>NVM_NV_BLOCK_NUM &lt; 2</strong>&lt;sup&gt;**NVM_DATASET_SELECTI&lt;/sup&gt;ON_BITS</th>
<th>(NATIVE,1,0)</th>
<th>(NATIVE,1,1)</th>
<th>(RENDUDANT,2,0)</th>
<th>(RENDUDANT,2,1)</th>
<th>(DATASET,1,0)</th>
<th>(DATASET,10,0)</th>
<th>(DATASET,11,0)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NVM WRITE_BLOCK_CRC</strong></td>
<td>TRUE</td>
<td>FALSE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>NVM RESISTANT TO CHANGED SW</strong></td>
<td>(NVM_RESISTANT_TO_CHANGED_SW eq TRUE)</td>
<td></td>
<td>(NVM_DYNAMIC_CONFIGURATION eq TRUE)</td>
<td>TRUE</td>
<td>FALSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>NVM BLOCK_JOB_PRIORITY</strong></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>255</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>NVM RAM BLOCK_LENGTH</strong></td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| **NVM_NV_BLOCK_DATA_ADDRESS** | RamAddress+N | NULL |
| **NVM_ROM_BLOCK_DATA_ADDRESS** | RomAddress+N | NULL |
| **NVM INIT_BLOCK_CALLBACK** | TestStubInitBlockCallback<n> | NULL |
| **NVM SINGLE_BLOCK_CALLBACK** | TestStubSingleBlockCallback<n> | NULL |
| **NVM SELECT_BLOCK FOR READALL** | TRUE | FALSE |

| **TCInitialRamBlockState** | VALID_INVALIDCRC1 | VALID_VALIDCRC | INVALID |
| **TCMemIfBehavior** | DO_NOT_ACCEPT_JOB | FAIL_JOB | CONTENTS_INVALID_ID | WRONG_CRC | SUCCESS |
| **TCMemIfBehaviorRedundantBlock** | DO_NOT_ACCEPT_JOB | FAIL_JOB | CONTENTS_INVALID_ID | WRONG_CRC | SUCCESS |

### [EXCLUSION]

- **NVM BLOCK USE_CRC** | FALSE
- **NVM USE_CRC** | TRUE
- **NVM SELECT BLOCK FOR READALL** | TRUE
- **NVM BLOCK MANAGEMENT_TYPE** | DATASET

### [EXCLUSION]

- **NVM SELECT BLOCK FOR READALL** | NULL
- **NVM ROM BLOCK DATA ADDRESS** | NULL

### [EXCLUSION]

- **NVM SELECT BLOCK FOR READALL** | TRUE
- **NVM BLOCK_JOB_PRIORITY** | 0
- **NVM ROM BLOCK_LENGTH** | 1
- **NVM ROM BLOCK_DATA_ADDRESS** | NULL

### [EXCLUSION]

- **NVM BLOCK_USE_CRC** | FALSE
- **NVM_RESISTANT_TO_CHANGED_SW** | TRUE
- **NVM_NV_BLOCK_DATA_ADDRESS** | NULL
- **NVM_ROM_BLOCK_DATA_ADDRESS** | NULL

---

**Figure 12** – Example: Combination table for block-specific NvM configuration parameters
4.4 Preparation of Test Case Specification

As preparation for the test case specification (i.e. implementation) phase, the TTCN-3 API module (containing the BSW module’s data type definitions and API functions) and the TTCN-3 Config module (containing configuration and test parameters) are created. These two TTCN-3 modules are created automatically by two tools (see Figure 3):

<table>
<thead>
<tr>
<th>Converter</th>
<th>This tool converts the BSW module’s API functions and data types into the appropriate TTCN-3 syntax.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Combiner</td>
<td>This tool combines the definitions and rules in the combination table (see Chapter 4.3.3.3) and produces configuration sets in both AUTOSAR XML and TTCN-3 formats.</td>
</tr>
</tbody>
</table>
5 Implementation Phase

Two sets of products are developed in the implementation phase:

- Dynamic test cases and configuration data in TTCN-3
- Simulating BSW modules for validating test specifications

While the BSW module simulation is based solely on the refined SWS document, the implementation of the TTCN-3 test cases has the following inputs:

- Test objectives and test procedures for each test case in semi-formal notation
- Signatures of BSW module operations in TTCN-3
- BSW module configuration and test data (test case parameterization)
- UML model specifying the test architecture (test drivers, stubs, interfaces)

5.1 BSW Module Simulation

The BSW module simulation fakes the module’s externally observable and controllable behavior. The simulation is not necessarily efficient and makes certain assumptions to reduce implementation effort. For example, SWS requirements related to internal mechanisms (implementation details) that were stated for performance reasons can be ignored as long as the module’s functional behavior is not affected.

Since the BSW module simulation does not run on an embedded system, the following simplifications can be applied:

<table>
<thead>
<tr>
<th>Dynamic memory management</th>
<th>Embedded systems usually treat memory as static and the (maximum) memory size for data structures must be defined during coding. Dynamic memory management offers the programmer additional possibilities as memory can be allocated and released as needed at runtime.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simplified execution model</td>
<td>Ensuring reentrancy (when required) is a major difficulty in implementing embedded systems. The simulation uses a simplified model (i.e. sequential function execution). Reentrancy and race conditions for shared resources thus become “non-issues”.</td>
</tr>
<tr>
<td>Complex data structures</td>
<td>Embedded systems must handle their resources (i.e. CPU and memory) economically and complex data structures, possibly based on dynamic memory management, are often prohibited. The simulation can use them, however.</td>
</tr>
<tr>
<td>Advanced function libraries</td>
<td>Function libraries that implement commonly used functionality (e.g. queues) can be used. This can also involve “language extensions” implemented in libraries such as System-C.</td>
</tr>
<tr>
<td>Advanced design methodologies</td>
<td>Advanced design methodologies, such as object orientation, and their supporting tools (CASE) can be used.</td>
</tr>
</tbody>
</table>

In summary: simulations can be implemented by applying high-level, state-of-the-art software-engineering methods. In general, embedded system constraints need not be considered. However, the simulating nature of a BSW module double must not
impair the capability of a conformance test suite to test production implementations of BSW modules on targets (ECUs) and hosts.

### 5.2 Dynamic Test Cases

Dynamic test cases are described by TTCN-3 files that, along with the appropriate TTCN-3 configuration file, can be executed against a SUT. The TTCN-3 test cases are implemented directly from the test purpose and test procedure (test steps). The TTCN-3 source code structure shall be the same for all CTSpecs to simplify handling and maintenance. The basic structure is specified in the following section.

### 5.3 File Structure

The test cases are structured into various TTCN-3 files. Common definitions & functions (e.g. standard types, DET and DEM interfaces) are put in files with fixed names, like:

- **std_types.ttcn**: Specifies AUTOSAR standard types.
- **det_dem.ttcn**: Specifies the Development Error Tracer (DET) and Diagnostic Event Manager (DEM) interfaces that conformance tests stub out for SUTs to report errors.

The BSW module specific files have the module’s official abbreviation (e.g. NvM for NVRAM Manager) as prefix and a suffix indicating the file type. The following files make up the conformance test cases:

- ***_test_suite.ttcn**: Control part to select/deselect test cases for execution and time-guard test case executions.
- ***_test_cases.ttcn**: Groups test cases by test item or test functionality, e.g. i/o-test cases and module state test cases. Each test case can call the base functions.
- ***_base_functions.ttcn**: Base functions that encapsulate commonly used test functionality.
- ***_test_architecture.ttcn**: Specifies the test components, the SUT’s interfaces and their interactions. Furthermore, it provides generic functions for preparing and finalizing test cases and handling failure behavior.
- ***_api.ttcn**
- ***_api_types.ttcn**
- ***.par**: SUT’s operation interface. Test cases stimulate and observe the SUT’s behavior at this interface.

Module configuration data and test data in TTCN-3.
6 Validation Phase

Several defects might occur when defining and implementing conformance tests, for example:

- Deviation from the BSW module specifications (SWS)
- Control errors:
  - Errors in the sequence of test stimulus and observation
  - Errors in expressions, e.g. in state or result checks determining the verdict of a test case
  - Errors in the use of control structures
  - Wrong API calls and callbacks
- Data errors:
  - Wrong parameters values in API calls
  - Wrong configuration or test parameter values
  - Structural errors in implementing the test architecture
- Typographical and syntactical defects

The validation phase therefore has the following objectives:

1. Tests cases must be correct with regard to the SWS.
2. Test cases must comply with the test steps as described in Chapter 4.2.3.
3. CTSpec must comply with standard TTCN-3 and defined coding rules.

The following verify and validate the CTSpecs with respect to these objectives:

1. Reviews of test procedures and TTCN-3 code reviews to verify that the test cases check specified behavior and correctly implement the test steps.
2. Code reviews to verify that the implemented TTCN-3 test cases adhere to TTCN-3 coding guidelines.
3. Compiling and linking with the TTCN-3 API and Config modules to ensure formal correctness and interface compatibility.
4. Execution of conformance tests against BSW module simulations to validate that the test cases correctly report “pass” and “fail”.

The first three activities are straightforward but the fourth needs further explanation.

6.1 Test Case Validation using a Simulation of the BSW Module

This section describes in more detail the concept of validating conformance test cases by executing them against a simulation of the BSW module under test.
6.1.1 Motivation

Validating test cases with simulation has the following advantages:

- The simulation focuses on the BSW module’s functionality as a whole and not on individual requirements. This avoids code duplication and thus reduces implementation effort compared to preparing test cases for the CTSpec.

- The simulation is implemented in a high-level programming language on a standard PC. This reduces the overall implementation effort compared to implementing on an embedded system.

- The personnel familiar with the BSW module’s specification can use their preferred high-level programming language (e.g. Java, C++). This accelerates implementation of the simulation and results in fewer errors.

- Code coverage in the simulation can be analyzed during execution of the conformance test cases. The functionalities within the BSW module that have not been covered by the conformance test cases can thus be identified.

6.1.2 Validation Setup

Figure 13 shows the setup for validating the conformance test cases against a BSW module simulation. The Test System is realized according to [4]. However the SUT is replaced by a “SUT Substitute” composed of

- the BSW module simulation
- the validation target adapter

![Figure 13 – Validation Setup](image-url)
6.1.3 Error Tracing during Validation

The module simulations used to validate the test cases shall all implement DET development error detection mechanisms as specified in [6]. This ensures consistent interfaces among the simulations and should ease their integration, if necessary.

6.2 The Validation Workshop

A “validation workshop” shall be held to validate the test cases by executing the CTSpec against the BSW module simulation. The following should attend:

- the Test Assessor
- the Test Designer
- the Test Implementer
- the Test Validation (Simulation) Implementer

Involving knowledgeable representatives from test case, simulation and test infrastructure development ensures that problems can be quickly identified and solved.

6.2.1 Preparation for the Validation Workshop

Before the workshop is held, the Test Assessor must develop the test infrastructure (i.e. test adapters, test tools on Test-PC, communication network etc.) and set it up. The Test Assessor must also make end-to-end communication tests using the API of the BSW module under test. These tests verify that communication events are correctly transferred between the test cases and the target adapter.

6.2.2 Conducting the Validation Workshop

The target test adapter (implemented by the Test Assessor) is first integrated with the BSW module simulation (implemented by the Test Validation Implementer). After that, the test cases can be compiled and validation can start with simple test cases. The subsequent activities depend on the problems and errors found during test case execution. The workshop participants must work together to identify the causes of problems and to define the right approaches to solving them. The issues identified and the solutions must be documented for traceability.

6.3 Results of the Validation Workshop

The validation workshop finally results in:

- an improved version of the BSW module simulation
- an improved version of the TTCN-3 test cases
- an issue list composed of generic issues and test case-specific issues
All issue list items must be resolved (by involving the SWS authors, if necessary) before the CTSpec can be regarded as “validated”.

6.4 Validation against Misbehavior

The validation activities, especially the validation workshop, focus on eliminating errors in both the module simulation and the test cases. The simulation’s objective is to emulate correct module behavior and verify that the test cases correctly report “pass”. This approach does not validate that the test cases correctly report “fail” when executed against a non-conformant implementation, however.

In theory, validation against misbehavior is far more complex than validation against correct behavior because the number of conceivable faults is much higher than the number of correct results for any given module implementation. Therefore, the confidence in the CTSpec gained through validation against misbehavior is inherently limited by the number of faults injected into the simulation.

Validation against misbehavior must be carried out after the CTSpec has been validated against correct behavior since confidence in the correctness of the simulation must be gained before errors are purposely injected into the simulation.

The steps to validate against misbehavior are as follows:

1. For each test case, identify possible BSW module implementation errors that should result in a “fail” verdict. These errors must be linked to the test cases to maintain traceability.

2. Implement these errors in the BSW module simulation. It must be possible to switch each error on or off individually.

3. Execute the CTSpec once per injected fault against the BSW module simulation. For each run, switch on a different fault.

4. Verify that for each CTSpec execution, only those test cases report “fail” that are associated with the errors that were switched on during execution.

Some tasks must be carried out manually (e.g. error identification and implementation) and some tasks can be automated (e.g. execution with different errors activated, verification of failed test cases).
The Test Designer must define the extent of validating against misbehavior (i.e. the number of errors injected into the simulation) based on the impacts of false positives on the quality of the test case implementation. In general, having sound test cases (justified, traceable to SWS and module variants, focused, independent, simple, and short) simplifies mapping test cases to module variants (configurations) and injected faults which in turn requires fewer test validation runs and raises the efficiency of these mutation tests.
7 Formal Descriptions of Test Cases

Test procedure descriptions are used as a means of communication between the test case design and formalized test case implementations in TTCN-3. This chapter works out some details on how AUTOSAR conformance tests use TTCN-3.

7.1 Data Type Mapping

AUTOSAR has defined basic data types (see [5]) to be used within BSW module source code. These are then further mapped to C data types in order to account for particular target system properties (e.g. CPU type) and to optimize execution time.

Since conformance tests are independent of the target platform, only the basic AUTOSAR data types must be converted to TTCN-3 data types. The System and Platform Adapters must convert the TTCN-3 data types to the specific target platform data types (see [4]). In addition to the mapping of basic AUTOSAR data types, the mapping of enumeration data types and data type compositions has to be defined. C pointers are handled differently, however. All C pointer data types are mapped to the generic TTCN-3 PTRType data type.

7.1.1 Guideline

Generally the AUTOSAR BSW module data types should be mapped as directly as possible to the TTCN-3 data types used in the CTSpecs. The TTCN-3 data type definitions should also use the equivalent AUTOSAR data type names. Applying this guideline to basic AUTOSAR data types (e.g. uint8) is generally straightforward as one mapping rule is usually obviously the most direct one. Complex and composed C data types (e.g. enumerations, struct data types) can be mapped to TTCN-3 data types in different ways. Here, specific mappings must be defined.

7.1.2 Mapping Rules

7.1.2.1 Basic AUTOSAR Data Types

Table 4 maps the basic AUTOSAR data types (taken from [5]) to TTCN-3 data types:

<table>
<thead>
<tr>
<th>AUTOSAR data type</th>
<th>Corresponding TTCN-3 data type definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>boolean (basic TTCN-3 data type)</td>
</tr>
<tr>
<td>uint8</td>
<td>type integer uint8 (0..255)</td>
</tr>
<tr>
<td>uint16</td>
<td>type integer uint16 (0..65535)</td>
</tr>
<tr>
<td>uint32</td>
<td>type integer uint32 (0..4294967295)</td>
</tr>
<tr>
<td>sint8</td>
<td>type integer sint8 (-128..127)</td>
</tr>
<tr>
<td>sint16</td>
<td>type integer sint16 (-32768..32767)</td>
</tr>
</tbody>
</table>
### Table 4 – Mapping between AUTOSAR data types and TTCN-3 data types

<table>
<thead>
<tr>
<th>AUTOSAR data type</th>
<th>Corresponding TTCN-3 data type definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>sint32</td>
<td>type integer sint32 (-2147483648..2147483647)</td>
</tr>
<tr>
<td>uint8_least</td>
<td>not relevant</td>
</tr>
<tr>
<td>uint16_least</td>
<td>not relevant</td>
</tr>
<tr>
<td>uint32_least</td>
<td>not relevant</td>
</tr>
<tr>
<td>sint8_least</td>
<td>not relevant</td>
</tr>
<tr>
<td>sint16_least</td>
<td>not relevant</td>
</tr>
<tr>
<td>sint32_least</td>
<td>not relevant</td>
</tr>
<tr>
<td>float32</td>
<td>type float float32 (-3.4E38..3.4E38)</td>
</tr>
<tr>
<td>float64</td>
<td>type float float64 (-1.7976931348623157E308..1.7976931348623157E308)</td>
</tr>
</tbody>
</table>

Note that the optimized AUTOSAR integer data types (*_least) must not be used in BSW module APIs (see SWS item PLATFORM032 in [5]). Therefore, they are not relevant to conformance test cases.

Mapping the basic AUTOSAR data types to TTCN-3 data types in this way makes it possible to use the same names for these data types in both the C source code and the TTCN-3 test cases.

#### 7.1.2.2 Indefinite AUTOSAR Data Types

Some AUTOSAR data types are indefinite due to unresolved dependencies, e.g. hardware dependencies: Eep_AddressType of type “uint8 ... uint32”. Three approaches are possible. The approach shall be chosen considering the recommendations given below.

**Use the most comprehensive data type**

Map the indefinite AUTOSAR data type to the most comprehensive data type possible (in the example above, map to uint32).

Adopting this approach shifts the issue of converting the comprehensive data type (e.g. uint32) to the currently valid data type (e.g. uint16) to the TTCN-3 Coder/Decoder (see [4]). The Coder/Decoder must be altered to the valid data type and this approach shall therefore be avoided.

**Use an indefinite data type**

Map the indefinite AUTOSAR data type to a compatible TTCN-3 data type that leaves room for the indefiniteness (in the example above, map to octetstring of length (1, 2, 4)).
This approach is quite elegant since the different AUTOSAR data types can be mapped to one flexible TTCN-3 data type. The different AUTOSAR data types must be compatible with the flexible TTCN-3 data type (i.e. there is an unambiguous mapping) so that the Coder/Decoder can convert between the AUTOSAR data types and the TTCN-3 data type.

However, the flexible TTCN-3 data type may not be compatible in all aspects. For example, the TTCN-3 octetstring of length (1, 2, 4) used to map the AUTOSAR uint8, uint16, uint32 data types cannot be used for arithmetic calculations directly. In this case, type conversion routines must be implemented before performing arithmetic operations.

Use a placeholder

Leave a placeholder for the mapping, e.g. “type integer indefinite_integer”. Later when the concrete data type for the indefinite AUTOSAR data type is known, replace the placeholder by the matching TTCN-3 data type according to Chapter 7.1.2.

7.1.2.3 AUTOSAR Enumeration Data Types

AUTOSAR enumeration values are either implemented directly as C enumerations (enum) or defined as constants (#define). In both cases, an integer value (unique within the enumeration type) is associated with the enumeration value. The integer value associated with the enumeration type in an SWS document is either stated explicitly or given by the enumeration sequence (counting starts at zero).

Due to this strong association between AUTOSAR enumeration value and integer value, AUTOSAR enumeration values are mapped to TTCN-3 enumerated types.

Example:

AUTOSAR enumeration definition:

```plaintext
MemIf_StatusType:
Type: Enum
Range: MEMIF_UNINIT
       MEMIF_IDLE
       MEMIF_BUSY
       MEMIF_BUSY_INTERNAL
```

Equivalent TTCN-3 definition:

```plaintext
type enumerated MemIf_StatusType // Assume C enum semantics
{
  MEMIF_UNINIT (0),
  MEMIF_IDLE (1),
  MEMIF_BUSY (2),
  MEMIF_BUSY_INTERNAL (3)
}
```

7.2 Configuration Mechanism

BSW conformance tests evaluate BSW module configuration parameters as specified in an SWS, along with test specific parameters. In the following a TTCN-3 mod-
ule is presented which encapsulates parameter accesses. In TTCN-3, module parameters can be used to provide external parameters to a test suite before start-up of the test system, that is, without the need to re-process test code for each modification of a parameter value. TTCN-3 files (*.par) contain values of configuration and test specific parameters per tested module variant. In general, files with parameter values can be generated from formal descriptions of configuration parameters (in SWS) and a module’s ICS. A test suite uses these parameter files and does not need to modify them.

7.2.1 Data types of Test Parameters

The rules for data type mapping described in Chapter 7.1 apply when converting AUTOSAR configuration parameter values into appropriate TTCN-3 definitions.

7.2.2 Format of Test Parameters

In general, several configuration sets with different combinations of test parameter values are needed for conformance tests, one at least for each tested BSW module variant. It is therefore crucial that the parameters are in a form that can easily be integrated in a conformance test suite.

In the following example, parameters are defined within one TTCN-3 module. Test cases in other TTCN-3 modules can import the test parameter module to access the parameter values. A module-specific parameter is defined as a constant (default value), holds for the complete BSW module, and can be overwritten externally by test system users upon test system execution.

Example:

```tcl
import from NvM_Variant { const c_DefaultMode }
modulepar
{
    boolean NVM_POLLING_MODE := c_DefaultMode; // Module-specific
    //...
}
```

Module-entity specific parameters, however, are defined per entity, e.g. per memory block or port pin. It must be possible to select entity-specific entities. Entity-specific parameters can be defined along with module-specific parameters within one TTCN-3 module with data accessor functions that return parameter values for a module-entity identified by an index.

Example:

```tcl
function f_NvM_ReturnBlock(in integer BlockId) // Index
return NvMBlockType
{
    var NvMBlockType BlockCfg; // Entity-specific
    select (BlockId)
    {
        case(0)
        {
```
7.3 Control Part

The control part executes all test cases belonging to a CTSpec and thus specifies the test case sequence. The test case itself evaluates whether a test case is actually valid for the currently active configuration set. Examples of control part code can be found in Part 1 “TTCN-3 Core Language” of [1].

7.4 Handling Open Implementations

Analysis of four BSW Software Specification documents\(^7\) during the CTSpec pilot project has shown that SWS requirements often leave the implementation of certain functionality open within certain limits.

From a conformance testing viewpoint, granting this degree of implementation freedom often dictates that the functionality be tested indirectly. For example, the strategy (in terms of step size etc.) for realizing the multi-step CRC calculation in the NVRAM Manager is left to the implementer. This means that the classic black-box test approach – stimulation, observation, evaluation – with single events is not applicable. Multiple events (e.g. multiple partial CRC calculations) must be observed instead and their conformance with certain conditions (e.g. partial CRC calculations cover the whole memory block) must be evaluated. This shall be realized by Test Case Stubs in the test cases, i.e. independent test components that provide services to the Test Case Clients which execute the main test steps.

7.5 Pointer Handling

AUTOSAR BSW module APIs use pointers heavily to realize **out or inout** parameters i.e. the parameter passes data back (e.g. `GetVersionInfo` API calls). This is a pass-by-reference mechanism, as the parameters actually contain the addresses of the data. Basically there are two ways to handle this functionality in TTCN-3.

---

\(^7\) NVRAM Manager, Memory Abstraction Interface, EEPROM Abstraction Interface and EEPROM Driver
1. Abstract from pointers

TTCN-3 supports the notion of `out` and `inout` parameters natively. Therefore, it does not require pointer mechanisms as in C to pass parameter data from the called function to the calling function. The "abstract from pointers" approach therefore defines the type of `out` or `inout` data directly within the TTCN-3 function signatures. In other words, a pass-by-value semantic is used in TTCN-3 functions that represent BSW module APIs.

2. Transparent pointer handling

The alternative approach is to handle pointers transparently in TTCN-3; i.e. defining a TTCN-3 user type representing C pointers (a pointer type). This type is then used in TTCN-3 function signatures representing BSW module APIs with pointer parameters. Calling these TTCN-3 functions then requires providing address values for these pointer type parameters. In this case, a pass-by-reference semantic is applied to the TTCN-3 functions.

Pointers shall be handled transparently for the following reasons:

- Abstracting pointer parameters would require performing memory transactions and data type conversions on the SUT Adapter and Platform Adapter level with associated functionality in the Target Adapter. These operations would need to be implemented in these adapters making them more dependent on the BSW module under test, i.e. a greater part of these adapters would need to be implemented specifically for the BSW module under test.

- Handling pointers transparently in the CTSpecs reduces the System and Platform Adapters’ (including the Target Adapter) dependency on the BSW module’s APIs and they can be reused for other BSW modules with less modification. This approach puts additional effort in test case implementation, as the memory transaction and data type conversion operations involved with pointer parameters must be handled in the test cases. However, these operations can be defined generically (see [4]) and encapsulated in base functions (see Chapter 5.3). Therefore, the advantage of the System and Platform Adapters being more generic outweighs the additional effort in test case implementation.

Figure 15 depicts the pointer handling concept used in the TTCN-3 conformance tests. A SUT operation’s parameter of type pointer is represented in TTCN-3 scripts as `PtrType`. In this way, a pointer’s base type (e.g. whether it points to `uint8` or `uint16`) is lost in the TTCN-3 domain. However, the advantage of this generic `PtrType` is its simplicity – no pointer type conversions are required. When a test case dereferences a pointer to something, the base type (structure) must be known anyway. This eliminates the need for typed pointers.
Operations on memory blocks for handling BSW module APIs with pointer parameters are covered in [4]. How pointers are handled depends on how the SUT handles memory addresses:

- For validation, a BSW module simulation is used as SUT. The pointer handling concept must be mapped to the simulation’s memory handling concept.
  - For a JAVA simulation, a mapping between the memory addresses (i.e. content of TTCN-3 pointer variables) and the JAVA objects that represent memory blocks has to be implemented in the target adapter.
  - A simulation implemented in C can make use of direct references in the TTCN-3 pointer variables, i.e. the TTCN-3 pointer variables contain the addresses of the associated memory blocks in the simulation.

- A C implementation with real memory blocks is usually used for Class A and Class B test setups. Here, TTCN-3 pointer variables refer directly to the associated memory blocks on the target system.
  - Configuration set constraints (e.g. “permanent RAM block” addresses in the NVRAM Manager configuration sets) may require additional mapping schemes between TTCN-3 pointer variables and the real target memory block addresses. These additional mapping schemes must then be implemented as module-specific target adapter functionality.

The Test Assessor must pay attention to the mapping between TTCN-3 pointer variables and the (real or simulated) target memory blocks when integrating the target
adapter with the BSW module under test. If necessary, the target adapter has to be modified to fulfill the requirements of the SUT.

### 7.6 Error Condition Handling

During test case execution, error conditions due to misbehavior or unexpected events occur primarily on the SUT side. While the BSW module under test generally reports error conditions to the test executable through DEM\(^8\) and DET\(^9\) API functions, error conditions in the target adapter must be reported explicitly to the test executable.

Since the target adapter mostly contains functionality that is triggered by the Test PC, TTCN-3 exception mechanisms are sufficient for reporting error conditions. The CTSpec shall define possible exceptions for each API and target adapter function with a common type (e.g. `uint8` values for error codes).

**Example:**

```plaintext
signature NvM_Init () exception (TA_ErrorType);
```

When it detects an internal error (e.g. out of memory), a target adapter should raise an exception with appropriate error code instead of replying to its invoking test executable. The test executable catches the exception, makes a test log entry and reports an error.

### 7.7 TTCN-3 Coding Style

TTCN-3 can be considered as a programming language. In large projects it is important to have the TTCN-3 well written. Hence, usual issues of how to write programs have to be taken into account and additionally issues of how to write compact and maintainable test code by means of TTCN-3. A style guide enables a common, consistent structure of all parts (generated and non-generated) of the test implementation. This style guide is similar to other style guides used during software development (see, for example, [8] Chapter 13). The style guide shall be created before the test implementation phase starts (see Figure 1).

---

\(^8\) Diagnostic Event Manager, a standardized AUTOSAR BSW module  
\(^9\) Development Error Tracer, a standardized AUTOSAR BSW module