Document Title	Explanation of Error Handling on Application Level
Document Category	Explanation (EXP)
Document UID	378
Document Status	published
Part of AUTOSAR Standard	Classic Platform
Part of Standard Release	R26-11

Document Change History

Date	Release	Changed by	Description
2026-11-26	R26-11	AUTOSAR Release Management	• No content changes
2025-11-27	R25-11	AUTOSAR Release Management	• No content changes
2024-11-27	R24-11	AUTOSAR Release Management	• Cleanup after removal of partition restart option
2023-11-23	R23-11	AUTOSAR Release Management	• Replaced symbols RESTART and NO_RESTART by OS_OSAPPLICATION_RESTART and OS_OSAPPLICATION_NO_RESTART.
2022-11-24	R22-11	AUTOSAR Release Management	• No content changes. • Editorial changes: – Document converted from Word to LaTeX. – Document structure adjusted to the standard.
2021-11-25	R21-11	AUTOSAR Release Management	• No content changes
2020-11-30	R20-11	AUTOSAR Release Management	• No content changes
2019-11-28	R19-11	AUTOSAR Release Management	• No content changes • Changed Document Status from Final to published

△

2018-10-31	4.4.0	AUTOSAR Release Management	• Editorial changes
2017-12-08	4.3.1	AUTOSAR Release Management	• Editorial changes
2016-11-30	4.3.0	AUTOSAR Release Management	• minor corrections / clarifications / editorial changes; For details please refer to the ChangeDocumentation
2015-07-31	4.2.2	AUTOSAR Release Management	• minor corrections / clarifications / editorial changes; For details please refer to the ChangeDocumentation
2014-10-31	4.2.1	AUTOSAR Release Management	• Editorial changes
2013-03-15	4.1.1	AUTOSAR Release Management	• Finalized for Release 4.1
2011-12-22	4.0.3	AUTOSAR Release Management	• Initial release

Disclaimer

This work (specification and/or software implementation) 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 work.

The material contained in this work is protected by copyright and other types of intellectual property rights. The commercial exploitation of the material contained in this work requires a license to such intellectual property rights.

This work 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 work may be utilized or reproduced, in any form or by any means, without permission in writing from the publisher.

The work has been developed for automotive applications only. It has neither been developed, nor tested for non-automotive applications.

The word AUTOSAR and the AUTOSAR logo are registered trademarks.

Table of Contents

1 Introduction 6
1.1 Scope . . . 6
1.2 Guide to this Document . . . 7
2 Definition of terms and acronyms 8
2.1 Basic dependability terms . . . 8
2.2 Fault Detection, Isolation and Recovery (FDIR) . . . 9
3 Related Documentation 11
3.1 Input documents & related standards and norms . . . 11
4 Error model 13
5 Error handling mechanisms 16
5.1 Overview . . . 16
5.2 Plausibility checks . . . 16
5.2.1 Description . . . 16
5.2.2 Applicability . . . 18
5.2.3 Application level vs. BSW . . . 18
5.2.4 AUTOSAR References . . . 19
5.3 Substitute Values . . . 19
5.3.1 Description . . . 19
5.3.2 Applicability . . . 20
5.3.3 Application level vs. BSW . . . 20
5.3.4 AUTOSAR References . . . 20
5.4 Voting . . . 21
5.4.1 Description . . . 21
5.4.2 Applicability . . . 21
5.4.3 Application level vs. BSW . . . 22
5.4.4 AUTOSAR References . . . 22
5.5 Agreement . . . 22
5.5.1 Description . . . 22
5.5.2 Applicability . . . 23
5.5.3 Application level vs. BSW . . . 23
5.5.4 AUTOSAR References . . . 23
5.6 Checksums/Codes . . . 23
5.6.1 Description . . . 23
5.6.2 Applicability . . . 24
5.6.3 Application level vs. BSW . . . 24
5.6.4 AUTOSAR References . . . 24
5.7 Execution sequence monitoring . . . 25
5.7.1 Description . . . 25
5.7.2 Applicability . . . 26

5.7.3 Application level vs. BSW . . . 26
5.7.4 AUTOSAR References . . . 26
5.8 Aliveness monitoring . . . 27
5.8.1 Description . . . 27
5.8.2 Applicability . . . 27
5.8.3 Application level vs. BSW . . . 27
5.8.4 AUTOSAR References . . . 28
5.9 Status and Mode Management . . . 28
5.9.1 Description . . . 28
5.9.2 Applicability . . . 29
5.9.3 Application level vs. BSW . . . 30
5.9.4 AUTOSAR References . . . 30
5.10 Reconfiguration . . . 30
5.10.1 Description . . . 30
5.10.2 Applicability . . . 31
5.10.3 Application level vs. BSW . . . 31
5.10.4 AUTOSAR References . . . 32
5.11 Reset . . . 32
5.11.1 Description . . . 32
5.11.2 Applicability . . . 33
5.11.3 Application level vs. BSW . . . 33
5.11.4 AUTOSAR References . . . 34
5.12 Error Filtering . . . 35
5.12.1 Description . . . 35
5.12.2 Applicability . . . 35
5.12.3 Application level vs. BSW . . . 35
5.12.4 AUTOSAR References . . . 36
5.13 Memory Protection . . . 36
5.13.1 Description . . . 36
5.13.2 Applicability . . . 36
5.13.3 Application level vs. BSW . . . 36
5.13.4 AUTOSAR References . . . 37
5.14 Timing Protection . . . 37
5.14.1 Description . . . 37
5.14.2 Applicability . . . 37
5.14.3 Application level vs. BSW . . . 38
5.14.4 AUTOSAR References . . . 38
6 Aspect mapping 40
6.1 Overview . . . 40
6.2 Mapping to FDIR process and Error Model . . . 40
6.3 Mapping to implementation level . . . 42

1 Introduction

The purpose and aim of this document is to survey application level error handling mechanisms common in the automotive industry and available for use with AUTOSAR. This includes both handling of errors at the application level and handling of application level errors.

Error handling in this context refers to the complete handling chain, i.e., detection, isolation/identification and recovery. A set of error handling mechanisms useful for automotive systems is presented, which cover all three phases of error handling. Each mechanism is first described in a high-level manner, describing applicability for error handling and technical aspects. Then, AUTOSAR functionality related to the mechanism is reviewed and it is detailed where in an AUTOSAR system the mechanism is implemented or supported. Consequently, the list of mechanisms includes both mechanisms fully (or partially) supplied by AUTOSAR and mechanisms that should be implemented at SW-C-level by application developers, if incorporated into the system. Note that the set of mechanisms covered is not complete and limited to mechanisms that can be implemented for systems built on AUTOSAR release 4.0. Alternate and additional mechanisms are possible and future releases of AUTOSAR may enable even more error handling functionality. Also, this document does not prescribe the use of any mechanisms - the decision is of course solely up to the application developers and integrators.

This document is intended as a description of possible mechanisms and is primarily aimed at application/SW-C developers. However, it can also be of use to developers of BSW-modules. Focus is on random faults, not on systematic design faults (such as SW bugs). Examples of such faults include HW faults affecting the application, communication or peripheral devices. It focuses on errors most suitably handled by SW-Cs, not covering error handling within or below the RTE, such as COM and OS error handling.

1.1 Scope

This document concerns error handling from an application's point of view. That is, it describes mechanisms for detection, isolation and recovery at application level, as well as mechanisms that can handle faults relevant for applications (e.g. memory access violations or timing violations).

The focus is on handling of errors which mainly are the effects of random external faults. Even though systematic faults (i.e. design faults) can manifest themselves in the same way as external faults, these are not the primary target of this document. The handling of systematic faults is related to development (e.g., processes, design methodologies, and debugging) rather than error handling during system operation.

Error handling in AUTOSAR is not restricted to application level error handling. The BSW has a number of built-in error handling mechanisms which are able to provide

e.g. reliable communication, synchronization, etc. However, these mechanisms will not be described in this document.

1.2 Guide to this Document

Depending on the familiarity of the reader to the various terms and definitions used in the area of dependable systems, some parts of the document can be flipped through quite quickly or even skipped. If you are very familiar with the area of dependable system, you might even go to Section 6 directly. In Table 1.1 you will find a summary of the subsequent chapters in order to identify which parts are of most interest to you.

Section	Description
1.1 Scope	This section describes the assumptions made in this document. The assumptions concern for example existence of some basic dependability mechanisms in the BSW.
2 Definition of terms and acronyms	This section contains an overview of the terms used in this document, including descriptions of the terms fault, error, failure, a description of the FDIR (Fault Detection, Isolation and Recovery) process, and a description of various failure modes. If you are familiar with the concepts in the area of dependable systems, you can browse through this part rather quickly.
4 Error model	This section describes the types of errors that we have considered to be the most important ones from an automotive application point-of-view. The mechanisms listed in the subsequent chapters are all categorized according to their respective applicability to the handling of these errors.
5 Error handling mechanisms	This section lists and describes the error handling mechanisms provided or supported by AUTOSAR. for implementing application-level error handling. Each mechanism has a high-level description, a discussion on applicability, a discussion on implementation level (application vs. BSW) and an overview of the available AUTOSAR concepts and services that can be used for this type of mechanism.
6 Aspect mapping	This section provides an overview of the presented mechanisms and the mapping of these to the FDIR process, error models and implementation level.

Table 1.1: Overview of the contents of the remainder of this document.

2 Definition of terms and acronyms

The table below includes acronyms and abbreviations relevant to this document which are not included in the [1, AUTOSAR glossary].

Abbreviation / Acronym:	Description:

Table 2.1: Acronyms and abbreviations used in the scope of this Document

2.1 Basic dependability terms

The fundamental concepts and terms of dependability used in this document are adopted directly from [2]. This section contains a short overview of the main terms and definitions used for dependable systems. It should be noted here that the word *system* is used in a very wide sense in this context. A system can denote anything from a single SW-C to a complete vehicle with multiple networks and ECUs. However, as the document is aimed at application level error handling, a *system* in the rest of the document should denote a software application, potentially consisting of multiple SW-Cs, possibly mapped over a set of (distributed) ECUs.

The term *dependability* is defined as "the trustworthiness of a system such that reliance can justifiably be placed on the service it provides". This means that a dependable system is one upon which the user (either human or non-human) can place its trust in that the services provided by the system are correct. The dependability of a system is characterized by a set of attributes, compromised by a set of impairments, and achieved and analyzed by a set of means.

The dependability attributes characterize, and profile, the dependability of a given system. Some examples of attributes are *availability*, *reliability*, *safety*, *confidentiality*, *integrity*, and *maintainability*.

During the construction and the operation of a system (here used in a wide sense - a system can be any bounded entity, such as an entire ECU or a single SW-C), events may occur which reduce the trustworthiness of the system by introducing faults into the system. A *fault* is a transient or permanent change of the system such that its integrity deviates from the expected correct integrity. During system operation, faults may prevent the system from providing its intended service. These faults may be from an internal source (such as software defects) or an external source (such as external disturbances or aging of components). The events that may reduce the dependability of a system are referred to as the impairments of dependability.

The mere presence of faults is, however, not sufficient to reduce the dependability of a system. A fault must be activated, i.e., the part of the system in which the fault is located must be exercised in some way during system operation (e.g., faulty code must be executed, defective memory locations must be read, etc.). If this happens, the result

may be an error. If a fault is viewed as a disease, an error can be said to be a symptom of that disease. An *error* is defined as an erroneous (soft) state in the system, i.e., the state is different from the state the system would have had if the fault had not been present. An error which is activated may cause other errors to occur in the system. This process is called error propagation.

If errors propagate beyond the system barrier, i.e., if they are visible to the environment of the system, the error transforms into a *failure*, which means that the system no longer provides its specified functionality.

The causality chain, fault error failure, is also recursive in nature. Thus, a failure of one system is perceived as a fault by the enclosing system (i.e. the former is a sub-system of the latter). For instance, a failure in a specific software component can be seen as a fault in the overall application (consisting of a set of SW-Cs).

Therefore, we get the following sequence:

$\ldots$ *failure* $\rightarrow$ *fault* $\rightarrow$ *error* $\rightarrow$ *failure* $\rightarrow$ *fault* $\ldots$

The methods used to achieve and analyze the dependability of a system are called the *means of dependability*. The purpose of this document is to document and investigate the means provided by AUTOSAR for the disposal of application developers for implementing error handling.

Note that this document covers mechanisms which are active during system operation. It does not cover means such as processes and methodologies for achieving functional safety, as these apply during system development rather than system operation.

2.2 Fault Detection, Isolation and Recovery (FDIR)

The process of handling faults during system operation is often referred to as FDIR, which stands for Fault Detection, Isolation, and Recovery.

Detection: The crucial first step in handling a fault is of course to become aware that it has occurred. Without this detection, no further activities can be performed. When it comes to detection, the original fault is often very hard to detect. What can be detected are the effects of the fault, that is, errors. These are detected by monitoring the state of the system.

Errors can manifest in different ways. The main manifestations are

1. erroneous values in the system (data errors),
2. erroneous execution time (timing errors),
3. erroneous sequence or execution order (program flow errors).
4. erroneous access to system resources, such as memory

Errors may propagate and generate consecutive faults which in turn may result in new errors, e.g., an erroneous data value is used as a pointer and causes a memory ac-

cess violation, which may create an erroneous value in another data value if a value is written to the erroneous memory location. Most mechanisms used for detection of errors allows the system to perform some action to find out more regarding the source of the error (isolation) and to issue corrective or compensating actions (recovery). Ideally, detection is done before the error has propagated any further, thus making it possible to stop further propagation. However, in most cases additional recovery actions are needed, such as stopping the offending component or reconfiguring to alternate functionality.

Isolation: Once a fault (or error) has been successfully detected, damage assessment and damage control needs to be performed, i.e. there is a need for isolation. During isolation, efforts are made to find the root cause for the erroneous state in the system, and information (e.g. regarding the spread and cause of the error) is gathered for subsequent use during error recovery. It may not always be possible (or practical) to find the root cause of the erroneous state. Note that isolation in this document refers to isolating the source of the errors, such that recovery is possible. It does not refer to isolation of specific components of a system with the purpose of stopping errors from propagating. In a sense, the word identification may have been a better choice, but as the commonly used word in descriptions of the FDIR process is isolation we will use it here.

Recovery: When the isolation is complete, recovery actions will be initiated. These actions aim at transferring the system into a controlled state, which can be a completely recovered state where nominal service is provided, or a safe degraded state where a limited or no service is provided. The better the isolation results are, the better the recovery actions can perform.

If recovery is not successful, a failure may occur, i.e., the system is in an uncontrolled state and its service is not defined.

3 Related Documentation

3.1 Input documents & related standards and norms

This document is related to many other documents published within AUTOSAR, especially those handled by the AUTOSAR Functional Safety team. The purpose of this document is not to replace any of these other documents, but to view the work done in other work packages from an application developer's point of view. Consequently there is a significant amount of overlap between this document and other documents, which shows the maturity reached within AUTOSAR.

For each mechanism a list of related AUTOSAR documentation is presented, which forms the explicit relation between this document and other AUTOSAR documents.

Information about functional safety mechanisms and measures is distributed throughout the AUTOSAR specification documents. Unless one knows how functional safety mechanisms are supported and where the necessary information is specifically located, it is difficult to evaluate how a safety-relevant system can be implemented using AUTOSAR efficiently.

The AUTOSAR document [3] summarizes the key points related to functional safety in AUTOSAR, explains how functional safety mechanisms and measures can be used and references the respective documents. Furthermore, it helps to establish a mapping between [4] requirements and AUTOSAR measures and mechanisms.

[1] Glossary
AUTOSAR_FO_TR_Glossary

[2] Basic concepts and taxonomy of dependable and secure computing

[3] Overview of Functional Safety Measures in AUTOSAR
AUTOSAR_CP_EXP_FunctionalSafetyMeasures

[4] ISO 26262:2018 Road vehicles -– Functional Safety
https://www.iso.org

[5] Specification of SW-C End-to-End Communication Protection Library
AUTOSAR_CP_SWS_E2ELibrary

[6] Specification of Flash Test
AUTOSAR_CP_SWS_FlashTest

[7] Specification of Core Test
AUTOSAR_CP_SWS_CoreTest

[8] Specification of RTE Software
AUTOSAR_CP_SWS_RTE

[9] Specification of Communication
AUTOSAR_CP_SWS_COM

[10] Software Component Template
AUTOSAR_CP_TPS_SoftwareComponentTemplate

[11] Specification of NVRAM Manager
AUTOSAR_CP_SWS_NVRAMManager

[12] Specification of Crypto Service Manager
AUTOSAR_CP_SWS_CryptoServiceManager

[13] Specification of CRC Library
AUTOSAR_CP_SWS_CRCLibrary

[14] Specification of Watchdog Manager
AUTOSAR_CP_SWS_WatchdogManager

[15] Specification of Communication Manager
AUTOSAR_CP_SWS_COMManager

[16] Specification of Basic Software Mode Manager
AUTOSAR_CP_SWS_BSWModeManager

[17] Specification of ECU State Manager
AUTOSAR_CP_SWS_ECUStateManager

[18] Specification of Operating System
AUTOSAR_CP_SWS_OS

[19] Specification of Function Inhibition Manager
AUTOSAR_CP_SWS_FunctionInhibitionManager

[20] Specification of Diagnostic Communication Manager
AUTOSAR_CP_SWS_DiagnosticCommunicationManager

[21] Specification of Diagnostic Event Manager
AUTOSAR_CP_SWS_DiagnosticEventManager

4 Error model

Design of dependable systems is based on a systematic analysis of potential faults (*fault models*) i.e., a set of postulated faults, derived from the operating environment of the system, which helps the designer or user to predict the consequences of these fault and define mechanisms for handling (detect, recover from etc.) those particular faults.

Faults can manifest on all levels of a system, from pure random HW faults (e.g., bit-flips) to SW (e.g., design defects) and faults in the interaction between components (e.g., incomplete interface specifications). Similarly faults are introduced throughout the design process (requirements, analysis, design, implementation, etc.). As this document is aimed at *application level* error handling focus is on errors that are expected to be handled by AUTOSAR SW components, either because application level knowledge is needed for the FDIR process, or because they have propagated from lower layers. Note that some errors of those types considered in this document can be handled by the BSW, but some may propagate to the application level and must therefore be handled there..

It is important to note that the focus is on handling of errors, which are the effects of faults. Even though design faults can manifest themselves in the same way as external faults, these are not the primary target of this document. The handling of design faults is related to debugging rather than error handling.

This document only considers error handling during operation. Techniques for fault-avoidance and fault-removal through rigorous or formalized development processes are not in scope.

The focus is mainly on random external faults, i.e., faults whose appearance can be modeled as a random process. However, this does not mean that the presented mechanisms cannot handle systematic faults, as the consequences of such faults (the errors) can manifest the same way as random faults. Both transient and permanent faults are considered, where some mechanisms are more suitable for one or the other.

As only SW mechanisms are considered in this documents it is actually *errors* that are detected, and not faults directly. Consequently the term *error model* will be used throughout the rest of the document instead of fault model. As errors are faults that have been activated and propagated, a single error can (theoretically) have many possible root causes, i.e., faults.

To simplify the discussion, error models have been divided into a number of broader error classes as seen from Table 4.1. These classes were chosen as they are easily mapped onto SW mechanisms. However, it is important to note that the error models are interrelated. An erroneous data value used in a branch instruction may propagate and become a program flow error, which may delay (or change) the output of the execution causing for instance a late response, i.e., a timing error.

Error Type	Description
Data	A *data error* is characterized by an erroneous value of a parameter, variable or message. The source of the error can be either internal (e.g., SW defect) or external (e.g., malfunctioning sensor, other faulty SW-component). Handling of data errors can break a causality chain that would lead to subsequent errors that are more complicated to handle, such as program flow or access violations.
Program flow	*Program flow errors* (also "control flow errors") manifest as actual program flows different than expected, possibly leading to missed, wrong or superfluous operations being carried out. The source of the program flow error can be both internal (SW defects) and external (data errors).
Access	For increased separation between executing components the system designer can partition the SW and restrict access to resources from the partition, e.g., memory access. When a component tries to access a resource in another partition without the proper access rights an *access violation* occurs. Access errors can be the result of a data or program flow error, e.g., an invalid program counter or pointer.
Timing	A communication (message, function invocation, etc.) is time critical when the delivery time has an effect on the correctness/usefulness of the communication. A *timing error* can be a message being delivered early, late or missing completely (omission). The last type of timing error, omission, is of special interest and is sometimes referred to as crash or fail-silent behaviour (note that it may be impossible to distinguish between crash, which is an uncontrolled state, and fail-silence, which is a controlled state). Timing errors also refer to execution time, where strict deadlines can be defined on how long a component is allowed access to the CPU.
Asymmetric	When errors propagate from one SW-component to another using some means of communication one differentiates between *symmetric* and *asymmetric errors*. In the symmetric case all receivers receive the same (erroneous) value. When the component can fail by sending *different* values (all of which may be valid) the error is said to be asymmetric. This error model is sometimes also referred to as the Byzantine model, which implies that no assumptions whatsoever are made on the behaviour of a malfunctioning component. Byzantine errors can only be detected by use of redundant components exchanging values to reach a common result.

Table 4.1: Considered error types.

Since the scope of the document is limited to errors handled at the application level, not all error types are considered for the mechanisms presented in Section 5. The following error types are **not** considered explicitly:

- **Communication Errors**: These errors are not included since it is assumed that reliable communication is available to SW-Cs. The only communication errors possible would then be caused by design faults, i.e., bugs in SW-Cs. Note that SW-Cs are still expected to handle communication errors that are reported from COM, or COM is configured to handle the error in the BSW.
- **Deadlocks and livelocks**: Deadlock and livelock situations are detected by watchdog mechanisms in the BSW and are thus not considered further in this document. These situations can of course lead to other timing errors which can be detected at application level. In that case, applications can deal with the effects of deadlock or livelock situations, but not necessarily the root cause of these situations.
- **Occurring faults and errors in instruction code**: At application level, it is in general impossible to detect instructions which have become faulty as a result of a fault in the storage medium or the internals of the processor. However, such faults in most cases result in illegal instructions which are detected when the processor attempts to execute them. If the resulting instruction is a legal one, it is likely this will instead transform to other types of errors in the system (e.g. data errors, timing errors, etc.) which could then be detected and handled by

other means, such as those described in this document. In the BSW, there are components for testing the core of the processor, the flash memory and the RAM. These may detect anomalies which could lead to instruction code errors.

5 Error handling mechanisms

5.1 Overview

This section describes a number of mechanisms on a high, conceptual level. These mechanisms can be used by application developers to incorporate error handling in the description or in the realization of applications.

Each mechanism is classified as being applicable or not applicable to a certain error type (as defined in Section 4). If a mechanisms is *applicable* this indicates that the mechanism is suitable for use to detect (or isolate or recover from) a specific set of errors. It is important to note that it *does not* imply that all errors of a specific type are detected. Each mechanism needs to be tuned to detect the specific errors required, and may only be able to handle a subset of the errors in an error class.

Some mechanisms are only *partially applicable* if they can be used in a straightforward manner in conjunction with some other mechanisms. For instance, when detecting that the value received from a sensor is wrong a substitute (safe) value can be used instead as a form of recovery. However, this only partially solves the recovery from the failed sensor, additional mechanisms are needed as well to fully recover, e.g., re-initialization of sensor. Partially applicable mechanisms are also marked as applicable.

When a mechanism is *not applicable* it means that the mechanism does not have straightforward use for the specific step and error model. In some cases modifications could potentially give the mechanism some utility, but a better option most likely exists.

Note that some of the mechanisms listed below may have side-effects, such as memory access patterns and timing behaviour of the system. Especially the timing behaviour may be affected and this needs to be explicitly addressed by the designer, such that all timing characteristics of the system are known, both during normal operation and during error handling.

Note also that even if many mechanisms are described, it is not always needed to combine different mechanisms, and mechanisms may even interact badly with other mechanisms.

5.2 Plausibility checks

5.2.1 Description

One of the most common ways to incorporate application specific knowledge is to construct monitors which check that the current value of a variable or a set of variables maintains some predefined condition, i.e., that they are plausible. A plausibility check is a predicate defined over set of variables in the application that can be checked dynamically at run time. The values that are being checked may represent values used for calculations, state values, or other kinds of values. Plausibility checks come in

mainly two different flavours: i) checking the validity of a single value, and ii) comparing multiple values.

Checking the validity of a single value: As the name suggests, checks are made for the validity of a value, i.e., if the value falls in a range of "good" values, if it follows a known pattern or behaviour etc. Such discrimination can typically only be made using specific knowledge regarding the application and/or its environment, such as maximum vehicle speed, minimum engine temperature etc.

Checks can be of range type (within a range or set of "good" values) or of differential nature (change from previous value, less than etc). Differential changes can additionally be temporal, where the periodicity of the check can add additional conditions (speed cannot change more than x km/h within time T). Here further application specific information is needed such as periodicity of checks etc.

Comparison of multiple values: Checking the validity of a set of single values may bring that all values are valid. However, performing a comparison across all values in the set may reveal errors which otherwise would be missed, by detecting that a given combination of seemingly valid values is not plausible. These comparisons can be computed using physical relations between several values (e.g., engine speed compared with vehicle speed and gear ratio), or by comparing data from redundant sources (e.g. multiple temperature sensors measuring the same temperature). The main problem in this case is to discriminate the erroneous value from the correct values. For some applications (especially safety-related ones) redundant sources for data values can be used to increase confidence in data validity, for instance by using multiple sensor readings (e.g., redundant sensors or reading values twice). When two values exist a *comparison* can be made between the two values, where the result is either that they are identical (possibly within some tolerance margin) and thus deemed correct or dissimilar indicating an error (in one of the values). After error detection additional measures are needed for isolating and recovering from the error. Comparison as a mechanism differs from validity checks in that it is based solely on comparing two values, disregarding the plausibility of the values. Comparison differs from voting (Section 5.4) since it is handled within a single SW-C, whereas voting may span multiple SW-Cs, for instance by executing redundant SW-Cs (either diverse or identical) and voting on the outcome. Comparisons are made locally and are always binary, i.e., two values are compared

We choose to differentiate between plausibility checks and status checking. The latter can be made independent of application knowledge and is presented as a separate mechanism (Section 5.9). However, as a result of a failed plausibility check an application may set status flags which other application and BSW can check and act upon.

5.2.2 Applicability

Step\Error Model	Data	Program flow	Access	Timing	Asymmetric
Detection	X				
Isolation	X				
Recovery					

Table 5.1: Applicability matrix for *plausibility* checks.

Validity checks are used to detect data errors in an application, where ranges of allowed/disallowed values can be defined. It is limited to the designer's ability to define such ranges based on requirements and/or application knowledge. Maintenance and traceability of such ranges must be handled within the development process. Undocumented or not up-to-date checks remaining in production code present a risk for the dependability of the application.

In some cases "safe" values can be defined to be used instead of a value outside the range of valid values, which is defined as a separate mechanism (Section 5.3).

Validity checks are only useful for detecting data errors. Checks can in some cases be part of the isolation step, where additional information regarding an error is gained by use of additional plausibility checks, for instance by determining which value (out of several) is erroneous when comparison or voting (Section 5.4) is used. This way application specific knowledge can be used when isolating errors.

Comparison can detect data errors by identifying a discrepancy between multiple values. Since comparisons are based on data values no other error models are supported. It may be difficult to isolate which of the multiple values are the ones which are erroneous. If validity checks show no invalid values, it is not possible to indicate which value is erroneous.

Plausibility checks are not generally applicable for detecting program flow, timing or asymmetric errors.

Plausibility checks cannot generally be used for recovery.

5.2.3 Application level vs. BSW

Plausibility checks can be implemented as executable assertions, where the values of one or more variables are checked using simple if-statements. It is implemented in the source code of the SW-Component, but does not generally affect the overall structure of the application. Checks can in most cases be implemented with deterministic timing characteristics (not considering de-bouncing). Memory requirements for data are typically low, restricted to saving values for differential checks.

5.2.4 AUTOSAR References

Name	Type	Document	Comment
SW-C End to End Communication Protection	SWS	E2E Library [5]	Definition of protocols between sender and receiver.
Flash Test Core Test	SWS	FLSTST [6] CORTST [7]	Comparison with known signature.
Runtime Environment	SWS	RTE [8]	Supports range checks for scaled values.
AUTOSAR COM	SWS	COM [9]	PDU replication and comparison.

Table 5.2: AUTOSAR References for *plausibility checks*.

The SW-C End to End Communication Protection Library can be used to check whether a signal is coming from an unexpected SW-C sender, or when the received signal is providing the information that it is not supposed to provide. It also permits to define SW-Cs able to check whether a stream of instances of a signal have been received in sequence (depending on the use cases, this can also be configured at the AUTOSAR COM level).

The Flash Test and Core Test modules can be configured in the context of single processor or multi-processor ECUs to perform checks on the ECU's hardware. These checks are compared with known good signatures of the hardware.

The RTE can check if communicated values matches with their allowed range.

5.3 Substitute Values

5.3.1 Description

Once an error has been detected that would prohibit the correct value to be assigned to a signal, a substitute value may be assigned to that signal. This substitute value can then be used in subsequent calculations such that these render useful results, albeit possibly degraded in quality. Examples of situations where substitute values could be assigned are:

- A sensor is malfunctioning, or operating outside of its operating range (e.g., detected by a plausibility check), and the corresponding physical entity cannot be measured reliably. A substitute value can be assigned that will allow subsequent algorithms using this value to proceed with their calculations.
- The providing SW-C of an input signal was reported as malfunctioning and thus its results may not be trusted even though a particular value seems to be within its valid domain.
- Transitory checking (e.g., close time after boot sequence) necessary to fulfil plausibility check when sensors are not yet available. Generally, "pending" status flag is set.

Since a substitute value is used it may be useful to notify the receiver that the original value is not available using a signal qualifier. The receiver can then decide on how to interpret and use the value.

Note that there are no generic signal qualifier mechanisms in AUTOSAR. Applications should define their own mechanism, for example by transmitting a record with a value and a qualifier.

5.3.2 Applicability

Step\Error Model	Data	Program flow	Access	Timing	Asymmetric
Detection					
Isolation					
Recovery	X	X		X	X

Table 5.3: Applicability matrix for *substitute values.*

Assigning a substitute provides means for subsequent operations to proceed in a way that makes an end result useful. However, it will not provide recovery from an error as it is not alleviating the situation that lead to the erroneous state in first place. Thus, it is partially applicable to recovery as it allows an existing error to be masked to a certain degree.

As the erroneous state that is to be masked with substitute values can be the result of any type of underlying error, it is partially applicable to all types.

5.3.3 Application level vs. BSW

Substitute values often require application knowledge that is not present at platform level and assignment is in these cases performed in the SW-Cs. At configuration time BSW modules can be configured with default substitute values. However, application specific knowledge is still typically needed.

5.3.4 AUTOSAR References

Name	Type	Document	Comment
Software Component Template Runtime Environment AUTOSAR COM	TPS SWS	SWC-T [10] RTE [8] COM [9]	Usage of initial and default values.
Software Component Template NVRAM Manager	TPS SWS	SWC-T [10] NVM [11]	Usage of default ROM block or redundant block.

Table 5.4: AUTOSAR References for *substitute values*.

The SW-C designer can specify an `initValue` on the `UnqueuedReceiverComSpec`. This value is used when no values were received, but the application reads the value. It can also be used in case of invalid values, depending on the `UnqueuedReceiverComSpec`'s `handleInvalid` attribute (`dontInvalidate`, `keep`, `replace`). These `initValue`s are implemented by COM [9] or RTE [8] based on the SW-C XML description (see the Software Component Template [10]).

An `initValue` can also be specified for `UnqueuedSenderComSpec`, `InterRunnableVariables`, `PerInstanceMemory`, or the `ramBlocks` of `NvBlockComponentType`s (see the Software Component Template [10]).

The NVM [11] can use a default ROM block in case of failure. This block can be defined by the SW-C designer with a `ParameterDataPrototype` in the `defaultData` role (see `RoleBasedDataAssignment` in the Software Component Template [10]).

5.4 Voting

5.4.1 Description

A basic principle for building fault-tolerant systems is to execute fragments ("components") redundantly and then consolidate the results of each component by performing a vote on the results. The actual vote is typically performed by a dedicated component called "voter". Common voting algorithms include "simple majority", "2 out-of 3" etc.

Voting can be performed at multiple levels, from replicated runnables in one SW-C to application level voting across ECUs. Replication can be made on the binary/source code level or on the specification level. In the former case each component is a copy of the same original component, whereas in the latter case components are different, but are built using the same specification.

Contrary to the comparison mechanism presented in Section 5.2, voting can handle more than two replicas and, depending on the number of replicas voting also provide isolation (identification of faulty replica) and partial recovery (a good value is output).

5.4.2 Applicability

Step\Error Model	Data	Program flow	Access	Timing	Asymmetric
Detection	X				
Isolation	X				
Recovery	X				

Table 5.5: Applicability matrix for *voting*.

Voting is used to detect data errors. Furthermore, the source of the erroneous value can be identified if three or more values are voted upon (and at most one value is erroneous). As a correct value can be produced despite the presence of an error,

voting partially supports recovery. Additional means are needed to fully recover from the error.

5.4.3 Application level vs. BSW

Voting is performed in SW-Cs.

5.4.4 AUTOSAR References

AUTOSAR does not provide a voting service. Since voting mechanisms are not provided by the AUTOSAR BSW, these need to be implemented at application level in the SW-Cs who require them. AUTOSAR supports multiple instantiation of SW-Cs. This feature can be used by a SW-C implementer to implement a specific voting mechanism for an application.

5.5 Agreement

5.5.1 Description

When redundant components are used to increase the reliability of an application agreement may be needed for components (called participants) to *agree* on the value used (including the result of some computation) by exchanging the result of local computations as messages[1].

The difference between agreement and voting mechanisms is that when using agreement components interact to reach a decision, whereas in voting it is left to the voter to decide. Agreement protocols could be compared with closed loop systems, where the feedback consists of the sent messages received by all other parties. Voting can analogously be compared to an open loop system where the voter collects values and decides.

Agreement protocols can also handle asymmetric faults through multiple rounds of information exchange. Thus all (correct) participants agree on the same value as well as on the correctness of the other participants.

[1] Obviously other communication paradigms can be used, messages are used as illustration.

5.5.2 Applicability

Step\Error Model	Data	Program flow	Access	Timing	Asymmetric
Detection	X				X
Isolation	X				X
Recovery	X				X

Table 5.6: Applicability matrix for *agreement*.

As values are compared similar to voting, agreement mechanisms can detect and isolate data value errors. The extra rounds of information exchange allow also for detection and isolation of asymmetric errors. Agreement cannot handle either program flow or timing errors.

Recovery is partially supported in that a faulty participant can be identified and its behavior masked from affecting the system. Additional means are needed to fully recover from the error.

5.5.3 Application level vs. BSW

Basic services are implemented on BSW-level, but applications using agreement protocol must be aware of the fact they participate. Proposal of new values and adoption of agreed values are examples of situations where applications need to be aware of the protocol.

5.5.4 AUTOSAR References

No true agreement service exists in AUTOSAR. If specific agreement semantics are needed for application level communication it shall be implemented specifically for these applications.

5.6 Checksums/Codes

5.6.1 Description

A technique for increased data consistency is to add redundant information to the data values to protect. The extra information allows for detection of modifications of (parts of) the data, and in some cases even correction and restoration of the original data values. The cost is both in terms of performance (time to calculate and check the checksum, additional communication needs) and in terms of the additional memory requirements for storage. Extensions also include cryptographic algorithms providing digital signatures and encryption/decryption of data.

There are multiple uses of checksums/codes, including:

- Safely storing data in both volatile and non-volatile memory

- Dependable communication between SW-C, both inter- and intra- ECU
- Protecting data from tampering (data integrity) by unauthorized entities
- Sending and receiving of encrypted data across unsafe channels

Please note the difference between the first two cases, which are concerned with benign errors, whereas the latter two are concerned with malicious errors (attackers, intruders etc). However, the same mechanisms can often be used for multiple purposes.

Additional threats to data security include spoofing (pretending to be someone else), repudiation (denying a performed action), denial of service, and elevation of privileges. In general this document is focused on benign errors. However, these threats may be of importance for some applications.

5.6.2 Applicability

Step\Error Model	Data	Program flow	Access	Timing	Asymmetric
Detection	X				X
Isolation	X				X
Recovery	X				X

Table 5.7: Applicability matrix for *checksums/codes*.

Checksums and codes are mainly targeted at protection of data throughout the FDIR process. Depending on the type of code used, all steps of the process can be supported.

Codes are also used as part of certain protocols (e.g., agreement) for handling asymmetric faults.

5.6.3 Application level vs. BSW

Checksums and cryptographic libraries are implemented on BSW-level or as libraries, due to performance and portability reasons. The use of dedicated peripheral circuits further decreases the use of application level mechanisms.

5.6.4 AUTOSAR References

Name	Type	Document	Comment
Crypto Service Manager	SWS	CSM [12]	Access to cryptographic functions/hardware.
CRC Routines.	SWS	CRC [13]	CRC routines.

▽

△

Name	Type	Document	Comment
SW-C End to End Communication Protection	SWS	E2E Library [5]	Additional CRC added to signals by SW-Cs.

Table 5.8: AUTOSAR References for *checksums/codes*.

The CSM can be used by SW-Cs to compute cryptographic checksums or codes through a port interface.

The CRC is a library which can be used directly by SW-Cs to compute checksums.

The SW-C End-to-End communication protection library can be used to define a protocol and protect with a checksum or code the data sent by SW-C through port interfaces.

5.7 Execution sequence monitoring

5.7.1 Description

Correct execution of an application includes that the sequence of executable entities is correct. Monitoring execution sequence will enable the detection of erroneous execution paths that may result in erroneous results.

Monitoring of execution sequence can be performed at different levels of granularity. Some examples of levels of granularity are:

- **Individual statements**: This is the finest granularity with which execution sequence can be monitored at source code level. The sequence of individual statements in the code is monitored.
- **Basic blocks**: A basic block is a block of code that has exactly one entry point and one exit point and cannot be entered or exited outside of these two entry and exit points. Thus execution from entry point to exit point is strictly sequential. Note that the minimum basic block is a single statement. The execution sequence of basic blocks can be specified in a so called *control flow graph*, and monitoring execution sequence at this granularity would be to ensure that execution is performed according to this graph. Control flow in this context is synonymous to program flow.
- **Runnables**: A runnable has one entry point but may potentially have multiple exit points and several valid execution paths from entry point to these exit points. Also, the paths may include loops. At this level the execution sequence of the runnables of one (or more) application(s) is monitored.

Depending on the granularity of the monitor, the resource requirements may range from fairly low to very high. Monitoring the sequence of individual statements is likely to require huge amounts of memory and processing time, whereas the sequence of runnables could likely be monitored with very low overhead in memory and execution time.

5.7.2 Applicability

Step\Error Model	Data	Program flow	Access	Timing	Asymmetric
Detection		X			
Isolation					
Recovery					

Table 5.9: Applicability matrix for *execution sequence monitoring*.

Monitoring of execution sequence will detect errors in program flow. These errors may be the result of previous data errors, timing errors or asymmetric errors. However, this cannot be distinguished by the monitor itself.

A monitor can not be used to recover from an error as it only checks the current state against some predefined notion of correctness (this is true for all kinds of monitors, not only those for execution sequence).

5.7.3 Application level vs. BSW

The most practical approach is probably cooperation between applications and the BSW where the application provides the BSW with information on where it is in the execution trajectory and the BSW then checks whether this location is a valid one. This requires predefined valid trajectories. One approach could be to configure valid successors from a given location in the execution trajectory for a set of locations. The granularity (instruction, basic block, runnable) could then be defined at configuration time.

5.7.4 AUTOSAR References

Name	Type	Document	Comment
Watchdog Manager	SWS	WdgM [14]	Supervision counter and program flow monitoring.
SW-C End-to-End Communication Protection	SWS	E2E Library [5]	Data sequence control on messages.
AUTOSAR COM	SWS	COM [9]	Sequence counters for messages sent over the bus.

Table 5.10: AUTOSAR References for *execution sequence monitoring*.

The Watchdog Manager can monitor heartbeats from application components not only for time but also for sequence. The correct sequence of execution is configured by the developers. The configuration contains the definition of a set of checkpoints or spy points and for each such point a set of allowed successors. It is then the responsibility of the SW-Cs (i.e. the developers) to make sure that each checkpoint/spy point is reported to the Watchdog Manager, which then checks the execution for temporal as well as logical sequence. In case of error detection, the ordinary recovery capabilities of the Watchdog Manager are utilized.

AUTOSAR COM and the SW-C E2E Communication Protection Library permit to check whether a stream of instances of a signal have been received in sequence.

5.8 Aliveness monitoring

5.8.1 Description

Aliveness monitoring deals with checking whether entities in a system are alive and well, i.e., are running as expected, in terms of periodicity or execution instances. A common way of monitoring aliveness is to monitor heartbeats from the parts that shall be monitored. If the heartbeat is within a certain range (minimum and maximum pulse), the monitored entity is said to be alive and well.

This mechanism is complementary to execution time monitoring, and deals with arrival rates rather than the time spent in a calculation.

Aliveness monitoring could be done at several levels:

- **Application level**: For example, if the application wants to monitor its internal components and thus, the various SW-Cs of the application could provide heartbeat signals.
- **BSW level**: For example, the BSW could monitor that the application components on an ECU behave as specified (within the limitations of the used monitoring principle).

5.8.2 Applicability

Step\Error Model	Data	Program flow	Access	Timing	Asymmetric
Detection				X	
Isolation					
Recovery					

Table 5.11: Applicability matrix for *aliveness monitoring*.

5.8.3 Application level vs. BSW

The heartbeats would have to be generated by the applications, but the checking of the pulse could be done in the BSW. Each stream of heartbeats can be configured for a certain pulse range (minimum and maximum thresholds) and ECU mode (Heartbeats depend from the ECU mode: boot, standby, etc.).

5.8.4 AUTOSAR References

Name	Type	Document	Comment
Watchdog Manager	SWS	WdgM [14]	The hub for collecting aliveness proofs and triggering the hw watchdogs through the Watchdog interface and drivers.

Table 5.12: AUTOSAR References for *aliveness monitoring*.

The watchdog manager can be used to supervise SW-Cs and/or BSW Modules. Which "entities" to supervise is pre-configured, together with the supervision parameters (no complete list, see Watchdog Manager SWS [14]):

- The expected number of *aliveness indications* within a certain amount of *supervision reference* cycles.
- Tolerance levels on the detection.
- The tolerable number of *failed reference* cycles

A supervised entity can be in one of three states, `WDGM_MONITORING_OK`, `WDGM_MONITORING_FAILED` or `WDGM_MONITORING_EXPIRED`. When the state changes from `WDGM_MONITORING_OK` to `WDGM_MONITORING_FAILED` recovery can be initiated and if successful (the number of aliveness indications has reached the tolerable limit before the number of tolerable *failed reference cycles* is reached) the watchdog is triggered and no actions are performed. When the number of allowed *failed reference cycles* is exceeded (the monitoring of the supervised entity has failed permanently), the state changes from `WDGM_MONITORING_FAILED` to `WDGM_MONITORING_EXPIRED` and the watchdog will not be triggered anymore.

5.9 Status and Mode Management

5.9.1 Description

Status and mode management deals with meta information for signals, applications, devices, etc. This meta information can be used to analyse the state of the system in order to isolate a faulty subsystem/component and modify its behaviour accordingly. This type of information is defined at various levels:

- **Signal status**: In addition to the value, a signal may have meta information associated with it, such as
  - **Signal quality**: This indicates the quality of the value, such as *nominal value*, *modelled value*, *replacement value*, and *default value*. The receiver of the signal may react differently depending on the quality of the signal.
  - **Signal timestamp**: This indicates when the value was created and can be used to check the age of a signal. If a calculation uses several input signals, one can check that all input signals are created within a tolerable time window.

  - **Signal sequence number**: It may be of interest to check that signal values are received in a certain order, and that no values or lost between reads. Also, sequence numbers can be used in a similar fashion as timestamps in that they allow a check that all values in a group are from a particular creation window (same sequence number, or with a minimum/maximum deviation).
  - **Update information**: A consumer of a signal may want to know whether a signal has been updated since the last read or not.

- **Device status**: An application may want to know the status of the devices it uses, such as sensors and actuators. If a device is not in normal operation, applications may want to choose to deliver some form of degraded service.
- **Application status/mode**: An application can also have a status, or a mode, which indicates the overall health or operating situation of the application. This status/mode can be used for recovery purposes, both internally in the said application and externally by other application.
- **Vehicle mode**: A vehicle may be in a number of different modes (e.g. normal operation, parked, limp-home) and the applications will have to behave accordingly.
- **ECU Mode**: An ECU may be in different states, such as sleeping, running, powered down, and transitional states between such states.

In order to provide support for this kind of status and mode management, it must be possible to set and get this information at application level (although there may only be one producer of a particular piece of status information there may be many consumers).

5.9.2 Applicability

Step\Error Model	Data	Program flow	Access	Timing	Asymmetric
Detection					
Isolation	X	X		X	X
Recovery					

Table 5.13: Applicability matrix for *status and mode management*.

From an error handling point-of-view, status and mode information can be used in the isolation phase. We have chosen to tag this mechanism as applicable for recovery as this mechanism in itself is not useable for recovery. Recovery can of course be triggered as a result of the status or mode on the monitored entity. However, the action of recovering itself is not part of this mechanism.

5.9.3 Application level vs. BSW

BSW modules manage meta information and distribute this information from producers to consumers. The definition, setting and getting of this information is done at application level though.

5.9.4 AUTOSAR References

Name	Type	Document	Comment
Communication Manager	SWS	ComM [15]	ComM handles the communication modes of the ECU and can trigger a shutdown of the bus if no communication is required. It also implicitly keeps the ECU alive (interactions with BswM).
Runtime Environment	SWS	RTE [8]	RTE Spec, including application mode management. Communication of modes.
Software Component Template	TPS	SWC-T [10]	Application modes defined. Modelling of the modes communication.
BSW Mode Manager	SWS	BswM [16]	Modes and transitions management.
ECU State Manager	SWS	EcuM [17]	The ECU State manager manages the state of a single ECU.
Operating System	SWS	OS [18]	States of OS Applications. OS has states for other OS objects (tasks, resources, etc.) but those are not directly accessed by applications.

Table 5.14: AUTOSAR References for *status and mode management*.

The RTE defines status for read data values, through the `Rte_Read`, `Rte_IStatus`. Status can be `RTE_E_OK`, `RTE_E_INVALID` or `RTE_E_MAX_AGE_EXCEEDED`. `RTE_E_INVALID` refers to an explicitly invalidated data value and `RTE_E_MAX_AGE_EXCEEDED` refers to an outdated data element.

The RTE allows a SW-C to specify mode, which can be use to execute or inhibit runnables. Modes can be defined for application specific purposes. This should be sufficient for error handling purposes as well.

The ECU State manager (and the Basic Software Mode Manager when it is used) manages the state of a single ECU.

5.10 Reconfiguration

5.10.1 Description

A technique for building fault-tolerant systems is to detect and isolate faults and then reconfigure the system to no longer use the faulty component, or to reconfigure to provide only a degraded set of services (or level of service).

Examples of reconfiguration strategies

- Isolating faulty components by hindering further communication. This could also include shutting down components selectively.
- Reconfiguration of protocol parameters, for instance voting algorithms, tolerance levels etc.
- Degraded functionality, such as providing only ABS and no ESP or a special "limp home" mode.

Reconfiguration is typically controlled using static policies, which are configured at system configuration time. The policies define when a reconfiguration is triggered, and how it is performed. Common triggers include error signals, as is the case for the Function Inhibition Manager (FIM) defined in AUTOSAR, which is triggered by messages from the DEM upon error. The FIM is limited to only informing an application of a request to inhibit parts of SW-Cs (so called "functionalities") and cannot actively inhibit anything or trigger a reconfiguration.

Note that there is a difference between mode management (Section 5.9: Status and Mode Management) and reconfiguration. Mode Management is an infrastructure to transfer information on states (i.e., modes) in the system, such that certain actions can be taken. These actions may be reconfiguration actions.

5.10.2 Applicability

Step\Error Model	Data	Program flow	Access	Timing	Asymmetric
Detection					
Isolation					
Recovery	X	X	X	X	X

Table 5.15: Applicability matrix for *reconfiguration*.

Reconfiguration of SW-Cs is part of the recovery step and cannot aid in either detection or isolation. It can possibly apply to any error, given that a reconfiguration policy is defined.

5.10.3 Application level vs. BSW

Reconfiguration can be performed at the application level and with support by the BSW services. Reconfiguration due to errors may require pure BSW support (such as Reset, Section 5.11).

5.10.4 AUTOSAR References

Name	Type	Document	Comment
Function Inhibition Manager	SWS	FIM [19]	The FIM can be used to selectively deactivate SW-C functionalities.
BSW Mode Manager	SWS	BswM [16]	The BswM arbitrates mode requests from SW-Cs in the application layer and performs mode switches based on pre-defined rules.
Operating System	SWS	OS [18]	The OS handles termination of OS-Applications.
Runtime Environment Software Component Template	SWS TPS	RTE [8] SWC-T [10]	Definition and support of modes. Definition and support for termination.

Table 5.16: AUTOSAR References for *reconfiguration*.

The FIM provides a control mechanism for SW-Cs and the functionality therein. In this context, a functionality can be built up of the contents of one, several or parts of runnable entities with the same set of permission/inhibit conditions. By means of the FIM, inhibiting (→ deactivation of application function) these functionalities can be configured and during runtime facilitating reconfiguration of the application.

The BswM can be configured to switch mode of the BSW based on mode requests. The interaction with SW-Cs is performed through the RTE using `ModeDeclarationGroup`s. The BswM thus performs two basic tasks: Mode Arbitration and Mode Control. The Mode Arbitration part initiates mode switches resulting from rule based arbitration of mode requests and mode indications received from SW-Cs or other BSW modules. The Mode Control part performs the mode switches by execution of action lists containing mode switch operations of other Basic Software modules. The action lists associated with a mode switch can be used to reconfigure the application, such as start/stop of I-PDUs (COM), disable all communication (NM) or changed PduR routing etc.

Termination/reconfiguration using the FIM requires cooperation with the affected SW-Cs implying correct behaviour from the SW-C. In case of malfunctioning SW-Cs a more brute force approach is required which terminates SW-Cs independently of the correctness of their behaviour.

The AUTOSAR OS handles termination of OS-Applications, to which tasks and other OS resources belong. The OS provides a service to terminate an OS-Application (`TerminateApplication`). This OS service can be called from an SW-C belonging to a trusted OS-Application.

5.11 Reset

5.11.1 Description

An application may try to recover in a number of ways, ranging from setting replacement values and wait, hoping that the error will disappear by itself (transient/intermittent errors) to full reconfiguration of its structure in order to shutdown faulty components

and launch backups and replacement components (permanent faults). Sometimes, it may not be possible to do these things though and a complete reset is necessary to start from a known good state. This holds not only for transient HW faults, but also for "soft" SW faults (systematic faults), sometimes referred to as Heisenbugs [8]. Such faults are transient in nature and cannot be easily repeated. A reset of the SW state typically removes Heisenbugs, as it puts the SW back in a known and well tested state.

Resets are used to recovery from transient faults. Permanent faults (e.g., permanently defect HW or SW bugs) cannot be recovered from. Other mechanisms to isolate transient faults are therefore needed to avoid using resets for permanent faults.

Reset can potentially be performed at the following levels:

- **SW-C reset**: An SW-C is found to be faulty and is reset in order to get it back into a safe state. The reset takes place at the application level.
- **Application reset**: If it is not sufficient to just reset single SW-Cs, it may be necessary to restart the whole application so that it can resume its normal service. The reset affects several SW-Cs at the application level, and may involve SW-Cs at multiple ECUs.
- **ECU reset**: If all else fails, it may be necessary to reset the entire ECU on which the fault or error has been found. This kind of reset will affect all applications that have SW-Cs located on the ECU as well as the BSW. The reset will also likely be visible to other ECU's on the network.

5.11.2 Applicability

Step\Error Model	Data	Program flow	Access	Timing	Asymmetric
Detection					
Isolation					
Recovery	X	X	X	X	X

Table 5.17: Applicability matrix for *reset*.

A reset is a last resort to recovery and is only applicable in that phase.

5.11.3 Application level vs. BSW

Reset at SW-C and application level could be controlled at application level, i.e., a dedicated SW-C could detect that an application has failed and request a reset. The RTE/OS shall provide services that enable the distribution of reset commands to the affected SW-Cs. For ECU reset, the BSW must be responsible for performing this. In the ECU case, all affected applications could be made aware of the impending reset in order to prepare themselves for it.

Note that even though the reset is initiated at the application level it will always require RTE/OS support for performing it.

5.11.4 AUTOSAR References

Name	Type	Document	Comment
Watchdog Manager	SWS	WdgM [14]	The hub for collecting aliveness proofs and triggering the hw watchdogs through the Watchdog interface and drivers.
Operating System	SWS	OS [18]	Provides the possibility to terminate OS-Applications. Controls and monitors timing behaviour of tasks.
ECU State Manager	SWS	EcuM [17]	An SW-C can select the shutdown target, i.e., which activity shall be performed after an EcuM controlled shutdown.
Diagnostic Communication Manager	SWS	DCM [20]	Diagnostic Communication Manager can initiate resets through MCU.

Table 5.18: AUTOSAR references for *reset*.

The ECU State Manager provides an interface to SW-Cs for selecting different shutdown targets, that is, what the ECU shall do when a shutdown is performed by the EcuM. The shutdown target can be either sleep, reset or off.

The WdgM monitors SW-Cs based on the aliveness indications made. The WdgM provides three mechanisms for the aliveness monitoring:

1. Supervised entities.
2. Temporal program flow monitoring.
3. logical program flow monitoring [14].

Missing aliveness proofs from applications can trigger a variety of actions by the WdgM:

- Inform the offending SW-C using the mode management mechanism in the RTE ("Local Failure Recovery").
- Inform the DEM. SW-Cs (the offending one and/or others) can then react upon DEM notifications. ("Global Failure Recovery").
- Termination of a partition using the `TerminateApplication` service provided by the OS.
- Indicate to the watchdog driver that it shall cease triggering the HW watchdog, eventually leading to an ECU reset.
- Directly resetting the ECU through the MCU.

The AUTOSAR OS provides the possibility to terminate an OS-Application, which is a set of OS resources (including tasks, i.e., SW-Cs). This can be triggered either by a protection violation (such as memory or timing) or manually by a trusted SW-C. The manual reset request from a SW-C makes it possible to reset even distributed applications in a coordinated fashion.

The DCM can perform an ECU reset upon a diagnostic request from an external diagnostic client (Tester). Such clients could potentially be vehicle-internal and would then

be able to request ECU resets based on the observed state of the vehicle. However, this is marginally at application level and will not be considered further here.

5.12 Error Filtering

5.12.1 Description

In some situations taking recovery actions due to errors, for instance transients, may cause more damage than it does good. Reacting to such errors may cause an over-reaction, where the recovery actions may put the system in a state where it is less safe than previously (for instance while restarting ECUs). In such cases a filtering of the errors may be needed before certain recovery actions are taken.

A common example is discrimination between transient and permanent errors using counters, where erroneous behavior increases the counter and correct behavior decreases it. When it reaches a specific threshold, the error is classified as permanent (a failure) and recovery is initiated.

5.12.2 Applicability

Step\Error Model	Data	Program flow	Access	Timing	Asymmetric
Detection					
Isolation	X	X		X	X
Recovery					

Table 5.19: Applicability matrix for *filtering*.

Error filtering is only applicable as a means for isolation. It requires additional detection mechanisms and can trigger recovery mechanisms. However, filtering of errors may alleviate the need for unnecessary recovery actions, and thus contribute also to a better recovery strategy by gaining information on the nature of the error.

5.12.3 Application level vs. BSW

A central debouncing mechanism is provided by the DEM within BSW. On the application level, application-specific error filtering can be applied, e.g., classifying transient errors. Such classification needs to be implemented by the SW-Cs.

5.12.4 AUTOSAR References

Name	Type	Document	Comment
Diagnostics Event Manager	SWS	DEM [21]	The DEM provides de-bouncing mechanisms to confirm errors.

Table 5.20: AUTOSAR references for *filtering*.

5.13 Memory Protection

5.13.1 Description

Memory protection is used to protect against errors propagating from one protection domain (partition) to another. Partitions are defined to form error confinement regions, where applications can be placed for mutual protection. Such protection enables separation between applications and thus enables multiple suppliers of SW-Cs to deliver SW for an ECU. This is important both for analyzing and enforcing safety issues.

5.13.2 Applicability

Step\Error Model	Data	Program flow	Access	Timing	Asymmetric
Detection			X		
Isolation[2]					
Recovery			X		

Table 5.21: Applicability matrix for *memory protection*.

Memory protection is mainly an error detection mechanism (for memory access errors). However, as the execution is halted before the write has been performed error propagation is confined. To achieve full recovery additional mechanisms are required, for instance to terminate or restart the execution of the offending runnable/task/SW-C/partition without endangering the execution of other applications (See Section 5.10 - Reconfiguration). Therefore memory protection only partially supports recovery from memory access violations.

5.13.3 Application level vs. BSW

Memory protection mechanisms are implemented in the BSW with HW support. To fully support recovery the RTE needs to also be aware of the protection mechanisms and act when applications are terminated. The application needs not be aware that it is running in a specific partition.

[2] See the definition of isolation in section 2.2. A memory protection mechanism can be used to stop the propagation of an error, but it cannot identify the source of the error.

Memory protection is not a mechanism used directly by SW-Cs, but it is configured by application developers/ECU integrators and is therefore relevant for application developers.

5.13.4 AUTOSAR References

Name	Type	Document	Comment
Operating System	SWS	OS [18]	Provides basic memory protection, the possibility to terminate OS-Applications, and is involved in communication across protection boundaries.
Runtime Environment	SWS	RTE [8]	The RTE is involved in the termination of partitions, and is involved in communication across protection boundaries by ensuring communication consistency.

Table 5.22: AUTOSAR references for *memory protection*.

The AUTOSAR OS together with the RTE implements the memory protection facilities in the system. The OS provides the fundamental protection mechanisms together with HW support and the OS and the RTE facilitate the communication across protection boundaries.

For an application developer it is important to know if the application is to be put in a partition for protection and what actions will be taken in case of protection violations.

5.14 Timing Protection

5.14.1 Description

Timing protection refers to protecting the system against activities requiring too much time to complete, such as an executing component taking too much execution time on the processor and thereby hindering the execution of other components, communication delays, peripheral units not responding in time, etc.

For activities, one may define time budgets providing an upper limit on how much time a given acitivy may use. For example, one may choose to set execution time budgets for components, or a maximum response times for communication.

5.14.2 Applicability

Step\Error Model	Data	Program flow	Access	Timing	Asymmetric
Detection				X	
Isolation					



△

Step\Error Model	Data	Program flow	Access	Timing	Asymmetric
Recovery				X	

Table 5.23: Applicability matrix for *timing protection*.

Execution time monitoring can be used to detect that a SW-C (actually a task) has exceeded its assigned execution time budget. It cannot detect timing errors in communication directly. Communication time-out monitoring can detect when a response is not received within the expected time.

To fully recover additional mechanisms are needed (reset, reconfiguration, etc.) and thus timing protection only partially supports recovery.

5.14.3 Application level vs. BSW

Timing protection is implemented in the BSW, i.e., the BSW performs the actual monitoring. Violations may be reported to SW-Cs. Execution time budgets and communication response deadlines are configured by application developers/ECU integrators and are therefore relevant for application developers.

5.14.4 AUTOSAR References

Name	Type	Document	Comment
Operating System	SWS	OS [18]	Provides the possibility to terminate OS-Applications. Monitors timing behaviour of tasks.
Runtime Environment	SWS	RTE [8]	The RTE is involved in the termination of partitions and ensures platform consistency. Timeout monitoring.
AUTOSAR COM	SWS	COM [9]	COM provides deadline monitoring for signals.
Watchdog Manager	SWS	WdgM [14]	The hub for collecting aliveness proofs and triggering the hw watchdogs through the Watchdog interface and drivers.
Software Component Template	TPS	SWC-T [10]	The SWC-T defines the requirements for the timeout handling, whether a SW-C supports restart, the Watchdog service needs.

Table 5.24: AUTOSAR references for *timing protection*.

The AUTOSAR OS provides basic timing protection facilities to monitor execution of tasks and ISRs. When a timing violation occurs the ECU-wide protection hook is called which has the possibility to terminate tasks or OS-Applications, shut down the OS or do nothing. As the scope for timing violation reactions can also be OS-Application wide, one can consider these mechanisms to act on partitions, and the developer/integrator can partition the system accordingly.

For an application developer it is important to know if the application is to be put in a partition for protection and what actions will be taken in case of protection violations.

Timeouts (called `aliveTimeout` in RTE [8]) can be defined for data elements exchanged using the RTE using sender-receiver communication. For communication on busses these correspond to signals for COM, which provides deadline monitoring, both for reception and transmission. Similarly, the RTE provides communication timeout monitoring for client-server communication. These types of deadline monitoring can be used to (for instance) detect that SW-Cs residing in terminated partitions no longer execute.

The WdgM monitors SW-Cs based on the aliveness indications made. The WdgM provides three mechanisms for the aliveness monitoring:

1. Supervised entities.
2. Temporal program flow monitoring.
3. Logical program flow monitoring.

See Section 5.11 and [14] for more information.

6 Aspect mapping

6.1 Overview

Each error handling mechanism is characterized by a number of properties, such as where in the FDIR process it applies or which error models it can handle. To give an overview of these different aspects of the different mechanism this section presents a number of mapping tables, where each mechanism is mapped onto the different properties. Each table gives references to the mechanisms covered individually in Section 5.

6.2 Mapping to FDIR process and Error Model

Not all mechanisms can be used for all steps in the FDIR process, and similarly they are applicable only for specific error models. To illustrate these 3-dimensional relationships (*Mechanism* x *FDIR step* x *Error Model*) we present three tables in this section. Each table shows an overview of the mechanisms and their applicability in each respective step of the FDIR process with respect to each of the error types defined in the error model.

More information regarding the capabilities of each error handling mechanism is found in Section 5, where each mechanism is presented in more detail.

In Table 6.1, the mechanisms are mapped to the first step of the FDIR process - *detection*, Table 6.2 contains the mapping to the second step - *isolation*, and Table 6.3 shows the mapping to the third and last step - *recovery*.

Mechanism	Ref	Data	Program flow	Access	Timing	Asymmetric
Plausibility checks	5.2	X				
Substitute values	5.3					
Voting	5.4	X				
Agreement	5.5	X				X
Checksums/Codes	5.6	X				
Execution sequence monitoring	5.7		X			
Aliveness monitoring	5.8				X	
Status & Mode Management	5.9					
Reconfiguration	5.10					
Reset	5.11					
Error filtering	5.12					
Memory protection	5.13			X		
Timing protection	5.14				X	

Table 6.1: Mapping of mechanisms to the steps of the FDIR process. Error detection.

Mechanism	Ref	Data	Program flow	Access	Timing	Asymmetric
Plausibility checks	5.2	X				
Substitute values	5.3					
Voting	5.4	X				
Agreement	5.5	X				X
Checksums/Codes	5.6					
Execution sequence monitoring	5.7					
Aliveness monitoring	5.8					
Status & Mode Management	5.9	X	X		X	X
Reconfiguration	5.10					
Reset	5.11					
Error Filtering	5.12	X	X		X	X
Memory protection	5.13					
Timing protection	5.14					

Table 6.2: Mapping of mechanisms to the steps of the FDIR process. Error isolation.

For error isolation it is important to note that the explicit information gained by detecting the error is not considered in Table 6.3. For example, when detecting that an entity has crashed by some aliveness monitoring mechanism the crashed entity is explicitly identified, however, no additional information is gained from the monitoring mechanism that can help in recovery, like the underlying reason for the crash. This is in contrast to for instance agreement on a data value, where not only the error is detected (some participant is faulty), but additionally also which participant.

The main purpose of status and mode management is the spread of error information, making it available to interested parties, and thereby making error isolation possible.

Mechanism	Ref	Data	Program flow	Access	Timing	Asymmetric
Plausibility checks	5.2					
Substitute values	5.3	X	X		X	X
Voting	5.4	X				
Agreement	5.5	X				X
Checksums/Codes	5.6	X				
Execution sequence monitoring	5.7					
Aliveness monitoring	5.8					
Status & Mode Management	5.9					
Reconfiguration	5.10	X	X	X	X	X
Reset	5.11	X	X	X	X	X
Error Filtering	5.12					
Memory protection	5.13			X		
Timing protection	5.14				X	

Table 6.3: Mapping of mechanisms to the steps of the FDIR process. Error recovery.

6.3 Mapping to implementation level

The implementation level refers to the level where the mechanism is most suitably implemented. However, the use and control of the mechanism is still in SW-Cs (or at least by application developers), i.e., on the application level.

Two implementation levels are relevant for the presented mechanisms, application level (SW-C) and basic SW level (BSW). The fundamental difference lies in where the mechanism is implemented, as a "service" provided to SW-Cs from the BSW level or as a pure application level mechanism, not requiring any specific BSW support. HW-based solutions are included at the BSW-level since direct access to HW is generally not permitted. As an example, dedicated cryptographic peripherals could be accessed through the same BSW interface as SW-based solutions.

Mechanism	Reference	Implementation level
Plausibility checks	5.2	SW-C
Substitute values	5.3	SW-C/BSW
Voting	5.4	SW-C
Agreement	5.5	SW-C
Checksums/Codes	5.6	SW-C/BSW
Execution sequence monitoring	5.7	SW-C/BSW
Aliveness monitoring	5.8	SW-C/BSW
Status & Mode Management	5.9	SW-C/BSW
Reconfiguration	5.10	SW-C/BSW
Reset	5.11	SW-C/BSW
Error Filtering	5.12	SW-C/BSW
Memory protection	5.13	BSW
Timing protection	5.14	BSW

Table 6.4: Mapping of mechanisms to implementation level.