Document Title	Specification of CRC Routines
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	16

Document Status	published
Part of AUTOSAR Standard	Classic Platform
Part of Standard Release	R21-11

Document Change History			
Date	**Release**	**Changed by**	**Description**
2021-11-25	R21-11	AUTOSAR Release Management	• Minor corrections / clarifications / editorial changes • Changed Document Status from Final to published
2020-11-30	R20-11	AUTOSAR Release Management	• Improved the structure of the 'error sections'
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	• Introduction of a new CRC-16 with the polynomial 0x8005 • Editorial changes
2017-12-08	4.3.1	AUTOSAR Release Management	• Editorial changes
2016-11-30	4.3.0	AUTOSAR Release Management	• Introduction of a new CRC-64 for E2E Profile 7 • Editorial changes
2015-07-31	4.2.2	AUTOSAR Release Management	• Corrected the magic check for the CRC32 and CRC32P4
2014-10-31	4.2.1	AUTOSAR Release Management	• Introduction of a new CRC-32 with the polynomial 0xF4ACFB13 • Editorial changes

2014-03-31	4.1.3	AUTOSAR Release Management	• CRC32 IEEE 802.3 check values corrected • Editorial changes
2013-10-31	4.1.2	AUTOSAR Release Management	• Editorial changes • Removed chapter(s) on change documentation
2013-03-15	4.1.1	AUTOSAR Administration	• New examples on how to use CRC routines and clarifications concerning CCITT standard • Removal of debugging concept
2011-12-22	4.0.3	AUTOSAR Administration	• The GetVersionInfo API is always available
2010-09-30	3.1.5	AUTOSAR Administration	• New parameter added to APIs in or-der to chain CRC computations • CRC check values corrected and checked values better explained • CRC magic check added
2010-02-02	3.1.4	AUTOSAR Administration	• Introduction of a new CRC-8 with the polynomial 2Fh • CRC-8 is now compliant to SAE J1850 • Legal disclaimer revised
2008-08-13	3.1.1	AUTOSAR Administration	• Separated CRC requirements from Memory Services Requirements • CRC8 management added
2008-02-01	3.0.2	AUTOSAR Administration	• Separated CRC requirements from Memory Services Requirements • CRC8 management added
2007-12-21	3.0.1	AUTOSAR Administration	• Document meta information extended • Small layout adaptations made
2007-01-24	2.1.15	AUTOSAR Administration	• "Advice for users" revised • "Revision Information" added
2006-11-28	2.1.1	AUTOSAR Administration	• `Crc_CalculateCRC16` and `Crc_CalculateCRC32` APIs, `Crc_DataPtr` parameter : void pointer changed to uint8 pointer • Legal disclaimer revised

2006-05-16	2.0	AUTOSAR Administration	• Document structure adapted to common Release 2.0 SWS Template • UML model introduction • Requirements traceability update • Reentrancy at calculating CRC with hardware support
2005-05-31	1.0	AUTOSAR Administration	• 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 and functional overview

This specification specifies the functionality, API and the configuration of the AUTOSAR Basic Software module CRC.

The CRC library contains the following routines for CRC calculation:

- CRC8: SAEJ1850

- CRC8H2F: CRC8 0x2F polynomial

- CRC16

- CRC32

- CRC32P4: CRC32 0xF4ACFB13 polynomial

- CRC64: CRC-64-ECMA

For all routines (CRC8, CRC8H2F, CRC16, CRC32, CRC32P4 and CRC64), the following calculation methods are possible:

- Table based calculation: Fast execution, but larger code size (ROM table)

- Runtime calculation: Slower execution, but small code size (no ROM table)

- Hardware supported CRC calculation (device specific): Fast execution, less CPU time

All routines are re-entrant and can be used by multiple applications at the same time. Hardware supported CRC calculation may be supported by some devices in the future.

2 Acronyms and Abbreviations

The glossary below includes acronyms and abbreviations relevant to the CRC module that are not included in the [1, AUTOSAR glossary].

Abbreviation / Acronym:	Description:
CRC	Cyclic Redundancy Check
ALU	Arithmetic Logic Unit

3 Related documentation

3.1 Input documents & related standards and norms

[1] Glossary
AUTOSAR_TR_Glossary

[2] General Specification of Basic Software Modules
AUTOSAR_SWS_BSWGeneral

[3] General Requirements on Basic Software Modules
AUTOSAR_SRS_BSWGeneral

[4] Requirements on Libraries
AUTOSAR_SRS_Libraries

[5] ITU-T Recommendation X.25: Interface between Data Terminal Equipment (DTE) and Data Circuit-terminating Equipment (DCE) for terminals operating in the packet mode and connected to public data networks by dedicated circuit
http://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-X.25-199610-I!!PDF-E&type=items

[6] 32-bit cyclic redundancy codes for Internet applications

[7] Listing of CRCs, including CRC-64-ECMA
https://en.wikipedia.org/wiki/Cyclic_redundancy_check

[8] Layered Software Architecture
AUTOSAR_EXP_LayeredSoftwareArchitecture

[9] Specification of ECU Configuration
AUTOSAR_TPS_ECUConfiguration

[10] List of Basic Software Modules
AUTOSAR_TR_BSWModuleList

3.2 Related specification

AUTOSAR provides a General Specification on Basic Software modules [2, SWS BSW General], which is also valid for CRC.

Thus, the specification SWS BSW General shall be considered as additional and required specification for AUTOSAR CRC Library.

4 Constraints and assumptions

4.1 Limitations

No known limitations.

4.2 Applicability to car domains

No restrictions.

5 Dependencies to other modules

There are no dependencies to other modules.

6 Requirements Tracing

The following tables reference the requirements specified in [3] and [4] and links to the fulfillment of these. Please note that if column "Satisfied by" is empty for a specific requirement this means that this requirement is not fulfilled by this document.

Requirement	Description	Satisfied by
[SRS_BSW_00005]	Modules of the μC Abstraction Layer (MCAL) may not have hard coded horizontal interfaces	[SWS_Crc_00051]
[SRS_BSW_00006]	The source code of software modules above the μC Abstraction Layer (MCAL) shall not be processor and compiler dependent.	[SWS_Crc_00051]
[SRS_BSW_00007]	All Basic SW Modules written in C language shall conform to the MISRA C 2012 Standard.	[SWS_Crc_00051]
[SRS_BSW_00009]	All Basic SW Modules shall be documented according to a common standard.	[SWS_Crc_00051]
[SRS_BSW_00010]	The memory consumption of all Basic SW Modules shall be documented for a defined configuration for all supported platforms.	[SWS_Crc_00051]
[SRS_BSW_00101]	The Basic Software Module shall be able to initialize variables and hardware in a separate initialization function	[SWS_Crc_00051]
[SRS_BSW_00160]	Configuration files of AUTOSAR Basic SW module shall be readable for human beings	[SWS_Crc_00051]
[SRS_BSW_00161]	The AUTOSAR Basic Software shall provide a microcontroller abstraction layer which provides a standardized interface to higher software layers	[SWS_Crc_00051]
[SRS_BSW_00162]	The AUTOSAR Basic Software shall provide a hardware abstraction layer	[SWS_Crc_00051]
[SRS_BSW_00164]	The Implementation of interrupt service routines shall be done by the Operating System, complex drivers or modules	[SWS_Crc_00051]
[SRS_BSW_00168]	SW components shall be tested by a function defined in a common API in the Basis-SW	[SWS_Crc_00051]
[SRS_BSW_00170]	The AUTOSAR SW Components shall provide information about their dependency from faults, signal qualities, driver demands	[SWS_Crc_00051]

Requirement	Description	Satisfied by
[SRS_BSW_00172]	The scheduling strategy that is built inside the Basic Software Modules shall be compatible with the strategy used in the system	[SWS_Crc_00051]
[SRS_BSW_00302]	All AUTOSAR Basic Software Modules shall only export information needed by other modules	[SWS_Crc_00051]
[SRS_BSW_00304]	All AUTOSAR Basic Software Modules shall use only AUTOSAR data types instead of native C data types	[SWS_Crc_00051]
[SRS_BSW_00305]	Data types naming convention	[SWS_Crc_00051]
[SRS_BSW_00306]	AUTOSAR Basic Software Modules shall be compiler and platform independent	[SWS_Crc_00051]
[SRS_BSW_00307]	Global variables naming convention	[SWS_Crc_00051]
[SRS_BSW_00308]	AUTOSAR Basic Software Modules shall not define global data in their header files, but in the C file	[SWS_Crc_00051]
[SRS_BSW_00309]	All AUTOSAR Basic Software Modules shall indicate all global data with read-only purposes by explicitly assigning the const keyword	[SWS_Crc_00051]
[SRS_BSW_00312]	Shared code shall be reentrant	[SWS_Crc_00051]
[SRS_BSW_00314]	All internal driver modules shall separate the interrupt frame definition from the service routine	[SWS_Crc_00051]
[SRS_BSW_00321]	The version numbers of AUTOSAR Basic Software Modules shall be enumerated according specific rules	[SWS_Crc_00051]
[SRS_BSW_00323]	All AUTOSAR Basic Software Modules shall check passed API parameters for validity	[SWS_Crc_00051]
[SRS_BSW_00325]	The runtime of interrupt service routines and functions that are running in interrupt context shall be kept short	[SWS_Crc_00051]
[SRS_BSW_00327]	Error values naming convention	[SWS_Crc_00051]
[SRS_BSW_00328]	All AUTOSAR Basic Software Modules shall avoid the duplication of code	[SWS_Crc_00051]
[SRS_BSW_00330]	It shall be allowed to use macros instead of functions where source code is used and runtime is critical	[SWS_Crc_00051]

Requirement	Description	Satisfied by
[SRS_BSW_00331]	All Basic Software Modules shall strictly separate error and status information	[SWS_Crc_00051]
[SRS_BSW_00333]	For each callback function it shall be specified if it is called from interrupt context or not	[SWS_Crc_00051]
[SRS_BSW_00334]	All Basic Software Modules shall provide an XML file that contains the meta data	[SWS_Crc_00051]
[SRS_BSW_00335]	Status values naming convention	[SWS_Crc_00051]
[SRS_BSW_00336]	Basic SW module shall be able to shutdown	[SWS_Crc_00051]
[SRS_BSW_00337]	Classification of development errors	[SWS_Crc_00051]
[SRS_BSW_00339]	Reporting of production relevant error status	[SWS_Crc_00051]
[SRS_BSW_00341]	Module documentation shall contains all needed informations	[SWS_Crc_00051]
[SRS_BSW_00342]	It shall be possible to create an AUTOSAR ECU out of modules provided as source code and modules provided as object code, even mixed	[SWS_Crc_00051]
[SRS_BSW_00343]	The unit of time for specification and configuration of Basic SW modules shall be preferably in physical time unit	[SWS_Crc_00051]
[SRS_BSW_00344]	BSW Modules shall support link-time configuration	[SWS_Crc_00051]
[SRS_BSW_00347]	A Naming seperation of different instances of BSW drivers shall be in place	[SWS_Crc_00051]
[SRS_BSW_00348]	All AUTOSAR standard types and constants shall be placed and organized in a standard type header file	[SWS_Crc_00051]
[SRS_BSW_00350]	All AUTOSAR Basic Software Modules shall allow the enabling/disabling of detection and reporting of development errors.	[SWS_Crc_00051]
[SRS_BSW_00353]	All integer type definitions of target and compiler specific scope shall be placed and organized in a single type header	[SWS_Crc_00051]
[SRS_BSW_00358]	The return type of init() functions implemented by AUTOSAR Basic Software Modules shall be void	[SWS_Crc_00051]

Requirement	Description	Satisfied by
[SRS_BSW_00359]	All AUTOSAR Basic Software Modules callback functions shall avoid return types other than void if possible	[SWS_Crc_00051]
[SRS_BSW_00360]	AUTOSAR Basic Software Modules callback functions are allowed to have parameters	[SWS_Crc_00051]
[SRS_BSW_00361]	All mappings of not standardized keywords of compiler specific scope shall be placed and organized in a compiler specific type and keyword header	[SWS_Crc_00051]
[SRS_BSW_00369]	All AUTOSAR Basic Software Modules shall not return specific development error codes via the API	[SWS_Crc_00051]
[SRS_BSW_00373]	The main processing function of each AUTOSAR Basic Software Module shall be named according the defined convention	[SWS_Crc_00051]
[SRS_BSW_00375]	Basic Software Modules shall report wake-up reasons	[SWS_Crc_00051]
[SRS_BSW_00378]	AUTOSAR shall provide a boolean type	[SWS_Crc_00051]
[SRS_BSW_00383]	The Basic Software Module specifications shall specify which other configuration files from other modules they use at least in the description	[SWS_Crc_00051]
[SRS_BSW_00384]	The Basic Software Module specifications shall specify at least in the description which other modules they require	[SWS_Crc_00051]
[SRS_BSW_00385]	List possible error notifications	[SWS_Crc_00051]
[SRS_BSW_00386]	The BSW shall specify the configuration for detecting an error	[SWS_Crc_00051]
[SRS_BSW_00388]	Containers shall be used to group configuration parameters that are defined for the same object	[SWS_Crc_00051]
[SRS_BSW_00389]	Containers shall have names	[SWS_Crc_00051]
[SRS_BSW_00395]	The Basic Software Module specifications shall list all configuration parameter dependencies	[SWS_Crc_00051]
[SRS_BSW_00398]	The link-time configuration is achieved on object code basis in the stage after compiling and before linking	[SWS_Crc_00051]
[SRS_BSW_00399]	Parameter-sets shall be located in a separate segment and shall be loaded after the code	[SWS_Crc_00051]

Requirement	Description	Satisfied by
[SRS_BSW_00400]	Parameter shall be selected from multiple sets of parameters after code has been loaded and started	[SWS_Crc_00051]
[SRS_BSW_00401]	Documentation of multiple instances of configuration parameters shall be available	[SWS_Crc_00051]
[SRS_BSW_00402]	Each module shall provide version information	[SWS_Crc_00050]
[SRS_BSW_00404]	BSW Modules shall support post-build configuration	[SWS_Crc_00051]
[SRS_BSW_00405]	BSW Modules shall support multiple configuration sets	[SWS_Crc_00051]
[SRS_BSW_00406]	A static status variable denoting if a BSW module is initialized shall be initialized with value 0 before any APIs of the BSW module is called	[SWS_Crc_00051]
[SRS_BSW_00407]	Each BSW module shall provide a function to read out the version information of a dedicated module implementation	[SWS_Crc_00011] [SWS_Crc_00017]
[SRS_BSW_00409]	All production code error ID symbols are defined by the Dem module and shall be retrieved by the other BSW modules from Dem configuration	[SWS_Crc_00051]
[SRS_BSW_00410]	Compiler switches shall have defined values	[SWS_Crc_00051]
[SRS_BSW_00411]	All AUTOSAR Basic Software Modules shall apply a naming rule for enabling/disabling the existence of the API	[SWS_Crc_00011] [SWS_Crc_00017]
[SRS_BSW_00414]	Init functions shall have a pointer to a configuration structure as single parameter	[SWS_Crc_00051]
[SRS_BSW_00415]	Interfaces which are provided exclusively for one module shall be separated into a dedicated header file	[SWS_Crc_00051]
[SRS_BSW_00416]	The sequence of modules to be initialized shall be configurable	[SWS_Crc_00051]
[SRS_BSW_00417]	Software which is not part of the SW-C shall report error events only after the DEM is fully operational.	[SWS_Crc_00051]
[SRS_BSW_00422]	Pre-de-bouncing of error status information is done within the DEM	[SWS_Crc_00051]
[SRS_BSW_00423]	BSW modules with AUTOSAR interfaces shall be describable with the means of the SW-C Template	[SWS_Crc_00051]

Requirement	Description	Satisfied by
[SRS_BSW_00424]	BSW module main processing functions shall not be allowed to enter a wait state	[SWS_Crc_00051]
[SRS_BSW_00425]	The BSW module description template shall provide means to model the defined trigger conditions of schedulable objects	[SWS_Crc_00051]
[SRS_BSW_00427]	ISR functions shall be defined and documented in the BSW module description template	[SWS_Crc_00051]
[SRS_BSW_00428]	A BSW module shall state if its main processing function(s) has to be executed in a specific order or sequence	[SWS_Crc_00051]
[SRS_BSW_00429]	Access to OS is restricted	[SWS_Crc_00051]
[SRS_BSW_00432]	Modules should have separate main processing functions for read/receive and write/transmit data path	[SWS_Crc_00051]
[SRS_BSW_00433]	Main processing functions are only allowed to be called from task bodies provided by the BSW Scheduler	[SWS_Crc_00051]
[SRS_LIBS_00001]	The functional behavior of each library functions shall not be configurable	[SWS_Crc_00051]
[SRS_LIBS_00002]	A library shall be operational before all BSW modules and application SW-Cs	[SWS_Crc_00051]
[SRS_LIBS_00003]	A library shall be operational until the shutdown	[SWS_Crc_00051]
[SRS_LIBS_00004]	Using libraries shall not pass through a port interface	[SWS_Crc_00051]
[SRS_LIBS_00005]	Each library shall provide one header file with its public interface	[SWS_Crc_00019] [SWS_Crc_00020] [SWS_Crc_00021] [SWS_Crc_00031] [SWS_Crc_00043] [SWS_Crc_00058] [SWS_Crc_00061] [SWS_Crc_00071]
[SRS_LIBS_00007]	Using a library should be documented	[SWS_Crc_00051]
[SRS_LIBS_00008]	For a given function prototype name, the behavior and the parameters shall not evolve once it is a part of an AUTOSAR final release	[SWS_Crc_00051]
[SRS_LIBS_00009]	All library functions shall be re-entrant	[SWS_Crc_00019] [SWS_Crc_00020] [SWS_Crc_00021] [SWS_Crc_00031] [SWS_Crc_00043] [SWS_Crc_00058] [SWS_Crc_00061] [SWS_Crc_00071]
[SRS_LIBS_00010]	A library shall define its own specific types in the library header file if and only if they are not yet defined by AUTOSAR	[SWS_Crc_00051]

Requirement	Description	Satisfied by
[SRS_LIBS_00011]	All function names and type names shall start with "Library short name_"	[SWS_Crc_00019] [SWS_Crc_00020] [SWS_Crc_00021] [SWS_Crc_00031] [SWS_Crc_00043] [SWS_Crc_00058] [SWS_Crc_00061] [SWS_Crc_00071]
[SRS_LIBS_00012]	Passing parameters with structure shall be allowed	[SWS_Crc_00051]
[SRS_LIBS_00013]	The error cases, resulting in the check at runtime of the value of input parameters, shall be listed in SWS	[SWS_Crc_00051]
[SRS_LIBS_00015]	It shall be possible to configure the microcontroller so that the library code is shared between all callers	[SWS_Crc_00051]
[SRS_LIBS_00016]	A SW-C may use a non-AUTOSAR library available on the market	[SWS_Crc_00051]
[SRS_LIBS_00017]	Usage of macros should be avoided	[SWS_Crc_00051]
[SRS_LIBS_00018]	A library function may only call library functions	[SWS_Crc_00072]
[SRS_LIBS_08518]	The CRC Library shall provide different calculation methods, optimizing either performance or memory usage	[SWS_Crc_00009] [SWS_Crc_00010] [SWS_Crc_00033] [SWS_Crc_00045] [SWS_Crc_00060] [SWS_Crc_00065] [SWS_Crc_00070]
[SRS_LIBS_08521]	All CRC routines shall allow step-by-step-wise calculation of a large data block	[SWS_Crc_00019] [SWS_Crc_00020] [SWS_Crc_00031] [SWS_Crc_00043] [SWS_Crc_00058] [SWS_Crc_00061] [SWS_Crc_00071]
[SRS_LIBS_08525]	The CRC library shall support the standard generator polynomials	[SWS_Crc_00002] [SWS_Crc_00003] [SWS_Crc_00015] [SWS_Crc_00016] [SWS_Crc_00030] [SWS_Crc_00032] [SWS_Crc_00042] [SWS_Crc_00044] [SWS_Crc_00052] [SWS_Crc_00053] [SWS_Crc_00054] [SWS_Crc_00055] [SWS_Crc_00056] [SWS_Crc_00057] [SWS_Crc_00059] [SWS_Crc_00062] [SWS_Crc_00063] [SWS_Crc_00064] [SWS_Crc_00067] [SWS_Crc_00068] [SWS_Crc_00069]
[SRS_LIBS_08526]	The CRC Library shall support current standards of CRC calculation	[SWS_Crc_00009] [SWS_Crc_00010] [SWS_Crc_00033] [SWS_Crc_00045] [SWS_Crc_00060] [SWS_Crc_00065] [SWS_Crc_00070]

7 Functional specification

7.1 Basic Concepts of CRC Codes

7.1.1 Mathematical Description

Let D be a bitwise representation of data with a total number of n bit, i.e.

$$D = (d_{n-1}, d_{n-2}, d_{n-3}, \ldots, d_1, d_0),$$

with $d_0, d_1, \ldots = 0b, 1b$. The corresponding Redundant Code C is represented by $n + k$ bit as

$$C = (D, R) = (d_{n-1}, d_{n-2}, d_{n-3}, \ldots, d_2, d_1, d_0, r_{k-1}, \ldots, r_2, r_1, r_0)$$

with $r_0, r_1, \ldots = 0b, 1b$ and $R = (r_{k-1}, \ldots, r_2, r_1, r_0)$. The code is simply a concatenation of the data and the redundant part. (For our application, we will chose k = 16, 32 and n as a multiple of 16 respectively 32).

CRC-Algorithms are related to *polynomials* with coefficients in the finite *field of two element*, using arithmetic operations $\oplus$ and $*$ according to the following tables.

The $\oplus$ operation is identified as the binary operation *exclusive-or*, that is usually available in the ALU of any CPU.

$\oplus$	$0b$	$1b$
$0b$	$0b$	$1b$
$1b$	$1b$	$0b$

$*$	$0b$	$1b$
$0b$	$0b$	$0b$
$1b$	$0b$	$1b$

For simplicity, we will write ab instead of $a * b$

We introduce some examples for *polynomials* with coefficients in the *field of two elements* and give the simplified notation of it.

(ex. 1) $p_1(X) = 1bX^3 + 0bX^2 + 1bX^1 + 0bX^0 = X^3 + X$

(ex. 2) $p_2(X) = 1bX^2 + 1bX^1 + 1bX^0 = X^2 + X^1 + 1b$

Any code word, represented by $n + k$ bit can be mapped to a polynomial of order

$n + k - 1$ with coefficients in the field of two elements. We use the intuitive mapping of the bits i.e.

$C(X) = d_{n-1}X^{k+n-1} + d_{n-2}X^{k+n-2} + \ldots + d_2 X^{k+2} + d_1 X^{k+1} + d_0 X^k + r_{k-1}X^{k-1} + r_{k-2}X^{k-2} + \ldots r_1 X + r_0$

$C(X) = X^k(d_{n-1}X^{n-1} + d_{n-2}X^{n-2} + \ldots + d_2 X^2 + d_1 X^1 + d_0) + r_{k-1}X^{k-1} + r_{k-2}X^{k-2} + \ldots + r_1 X + r_0$

$C(X) = X^k D(X) \oplus R(X)$

This mapping is one-to-one.

A certain space CRC_G of *Cyclic Redundant Code Polynomials* is defined to be a multiple of a given *Generator Polynomial* $G(X) = X^k + g_{k-1}X^{k-1} + g_{k-2}X^{k-2} + \ldots + g_2X^2 + g_1X + g_0$. By definition, for any code polynomial $C(X)$ in CRC_G there is a polynomial $M(X)$ with

$$C(X) = G(X)M(X)$$

For a fixed irreducible (i.e. prime-) polynomial $G(X)$, the mapping $M(X) \rightarrow C(X)$ is one-to-one. Now, how are data of a given codeword verified? This is basically a division of polynomials, using the *Euclidian Algorithm*. In practice, we are not interested in $M(X)$, but in the *remainder* of the division, $C(X) \bmod G(X)$. For a correct code word *C*, this remainder has to be *zero*, $C(X) \bmod G(X) = 0$. If this is not the case - there is an error in the codeword. Given $G(X)$ has some additional algebraic properties, one can determine the error-location and correct the codeword.

Calculating the code word from the data can also be done with the *Euclidian Algorithm*. For a given data polynomial $D(x) = d_{n-1}X^{n-1} + d_{n-2}X^{n-2} + \ldots + d_1X^1 + d_0$ and the corresponding code polynomial $C(X)$ we have

$$C(X) = X^kD(X) \oplus R(X) = M(X)G(X)$$

Performing the operation $\bmod G(X)$ on both sides, one obtains

$$0 = C(X) \bmod G(X) = [X^kD(X)] \bmod G(X) \oplus R(X) \bmod G(X) \qquad (7.1)$$

We denote that the order of the Polynomial R(X) is less than the order of G(X), so the modulo division gives zero with remainder $R(X)$:

$$R(X) \bmod G(X) = R(X)$$

For polynomial $R(X)$ with coefficients in the finite field with two elements we have the remarkable property $R(X) + R(X) = 0$. If we add $R(X)$ on both sides of equation 7.1 we obtain

$$R(X) = X^kD(X) \bmod G(X)$$

The important implication is that the redundant part of the requested code can be determined by using the Euclidian Algorithm for polynomials. At present, any CRC calculation method is a more or less sophisticated variation of this basic algorithm.

Up to this point, the propositions on CRC Codes are summarized as follows:

1. The construction principle of CRC Codes is based on polynomials with coefficients in the finite field of two elements. The $\oplus$ operation of this field is identical to the binary operation "XOR" (exclusive or)

2. There is a natural mapping of bit-sequences into this space of polynomials.

3. Both calculation and verification of the CRC code polynomial is based on division modulo a given generator polynomial.

4. This generator polynomial has to have certain algebraic properties in order to achieve error-detection and eventually error-correction.

7.1.2 Euclidian Algorithm for Binary Polynomials and Bit-Sequences

Given a Polynomial $P_n(X) = p_n X^n + p_{n-1} X^{n-1} + \ldots + p_2 X^2 + p_1 X + p_0$ with coefficients in the finite field of two elements. Let $Q(X) = X^k + q_{k-1} X^{k-1} + q_{k-2} X^{k-2} + \ldots + q_2 X^2 + q_1 X + q_0$ be another polynomial of exact order $k > 0$. Let $R_n(X)$ be the remainder of the polynomial division of maximum order $k - 1$ and $M_n(X)$ corresponding so that

$$R_n(X) \oplus M_n(X)Q(X) = P_n(X)$$

Euclidian Algorithm - Recursive

(Termination of recursion)

If $n < k$, then choose $R_n(X) = P_n(X)$ and $M_n = 0$.

(Recursion $n + 1 \rightarrow n$)

Let $P_{n+1}(X)$ be of maximum order $n + 1$.

If $n + 1 >= k$ calculate $P_n(X) = P_{n+1}(X) - p_{n+1}Q(X)X^{n-k+1}$. This polynomial is of maximum order n. Then

$$P_{n+1}(X) \bmod Q(X) = P_n(X) \bmod Q(X)$$

Proof of recursion

Choose $R_{n+1}(X) = P_{n+1}(X) \bmod Q(X)$ and $M_{n+1}(X)$ so that

$$R_{n+1}(X) \oplus M_{n+1}(X)Q(X) = P_{n+1}(X)$$

Then $R_{n+1}(X) - R_n(X) = P_{n+1}(X) - M_{n+1}(X)Q(X) - P_n(X) \oplus M_n(X)Q(X)$.

With $P_{n+1}(X) - P_n(X) = p_{n+1}Q(X)X^{n-k+1}$ we obtain:

$$R_{n+1}(X) - R_n(X) = p_{n+1}Q(X)X^{n-k+1} + M_n(X)Q(X) - M_{n+1}(X)Q(X)$$

$$R_{n+1}(X) - R_n(X) = Q(X)[p_{n+1}X^{n-k+1} + M_n(X) - M_{n+1}(X)]$$

On the left side, there is a polynomial of maximum order $k-1$. On the right side $Q(X)$ is of exact order k. This implies that both sides are trivial and equal to zero. One obtains

$$R_{n+1}(X) = R_n(X) \tag{7.2}$$

$$M_{n+1}(X) = M_n(X) + p_{n+1}X^{n-k+1} \tag{7.3}$$

(end of proof)

Example 7.1

$P(X) = P^4(X) = X^4 + X^2 + X + 1b; Q(X) = X^2 + X + 1b; n = 4; k = 2$

$P_3(X) = X^4 + X^2 + X + 1b - 1b(X^2 + X + 1b)X^2 = X^3 + X + 1b$

$P_2(X) = X^3 + X + 1b - 1bX(X^2 + X + 1b) = X^2 + 1b$

$P_1(X) = X^2 + 1 - 1b(X^2 + X + 1) = X$

$R(X) = P(X) \bmod Q(X) = R_1(X) = P_1(X) = X$

7.1.3 CRC calculation, Variations and Parameter

Based on the Euclidian Algorithm, some variations have been developed in order to improve the calculation performance. All these variations do not improve the capability to detect or correct errors - the so-called Hamming Distance of the resulting code is determined only by the generator polynomial. Variations simply optimize for different implementing ALUs.

CRC-Calculation methods are characterized as follows:

1. Rule for Mapping of Data to a bit sequence $(d_{n-1}, d_{n-2}, d_{n-3}, \ldots, d_1, d_0)$ and the corresponding data polynomial $D(X)$ (standard or reflected data).

2. Generator polynomial $G(X)$

3. Start value and corresponding Polynomial $S(X)$

4. Appendix $A(X)$, also called XOR-value for modifying the final result.

5. Rule for mapping the resulting CRC-remainder $R(X)$ to codeword. (Standard or reflected data)

The calculation itself is organized in the following steps

- Map Data to $D(X)$

- Perform Euclidian Algorithm on $X^k \, D(X) + X^{n-k-1}S(X) + A(X)$ and determine $R(X) = [X^k D(X) + X^{n-k-1}S(X) + A(X)] \bmod G(X)$

- Map $D(X), R(X)$ to codeword

7.2 Standard parameters

This section gives a rough overview on the standard parameters that are commonly used for 8-bit, 16-bit and 32-bit CRC calculation.

- CRC result width: Defines the result data width of the CRC calculation.

- Polynomial: Defines the generator polynomial which is used for the CRC algorithm.

- Initial value: Defines the start condition for the CRC algorithm.

- Input data reflected: Defines whether the bits of each input byte are reflected before being processed (see definition below).

- Result data reflected: Similar to "Input data reflected" this parameter defines whether the bits of the CRC result are reflected (see definition below). The result is reflected over 8-bit for a CRC8, over 16-bit for a CRC16 and over 32-bit for a CRC32.

- XOR value: This Value is XORed to the final register value before the value is returned as the official checksum.

- Check: This field is a check value that can be used as a weak validator of implementations of the algorithm. The field contains the checksum obtained when the ASCII values '1' '2' '3' '4' '5' '6' '7' '8' '9' corresponding to values 31h 32h 33h 34h 35h 36h 37h 38h 39h is fed through the specified algorithm.

- Magic check: The CRC checking process calculates the CRC over the entire data block, including the CRC result. An error-free data block will always result in the unique constant polynomial (magic check) - representing the CRC-result XORed with 'XOR value'- regardless of the data block content.

Example 7.2

Magic check calculation of SAE-J1850 CRC8 (see detailed parameters in [SWS_Crc_00030]) over data bytes 00h 00h 00h 00h:

- CRC generation: CRC over 00h 00h 00h 00h, start value FFh:

  - CRC-result = 59h

- CRC check: CRC over 00h 00h 00h 00h 59h, start value FFh:

  - CRC-result = 3Bh

  - Magic check = CRC-result XORed with 'XOR value': C4h = 3Bh xor FFh

Data reflection: It is a reflection on a bit basis where data bits are written in the reverse order. The formula is:

$$\text{reflect}_n(x) = \sum_{i=0}^{n-1} x_i \times 2^{n-i-1}$$

where x is the data and n the number of data bits.

E.g. The reflection$_{16}$ of 2D$_{16}$ (n = 8) (00101101$_2$) is B4$_{16}$ (10110100$_2$)

The reflection$_{16}$ of 12345678$_{16}$ (n = 16) (0001 0010 0011 0100 0101 0110 0111 1000$_2$) is 1E6A2C48$_{16}$ (0001 1110 0110 1010 0010 1100 0100 1000$_2$).

The reflection$_{32}$ of 123456789ABCDEF0 (n = 32) (0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111 0000$_2$) is 0F7B3D591E6A2C48$_{16}$ (0000 1111 0111 1011 0011 1101 0101 1001 0001 1110 0110 1010 0010 1100 0100 1000$_2$).

The reflection$_8$ of 123456789ABCDEF0 (n = 8) (0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111 0000$_2$) is 84C2A6E195D3B7F0$_{16}$ (1000 0100 1100 0010 1010 0110 1110 0001 1001 0101 1101 0011 1011 0111 1111 0000$_2$).

7.2.1 8-bit CRC calculation

7.2.1.1 8-bit SAE J1850 CRC Calculation

[SWS_Crc_00030] ⌈The `Crc_CalculateCRC8` function of the CRC module shall implement the CRC8 routine based on the SAE-J1850 CRC8 Standard:

CRC result width:	8 bits
Polynomial:	1Dh
Initial value:	FFh
Input data reflected:	No
Result data reflected:	No
XOR value:	FFh
Check:	4Bh
Magic check:	C4h

⌋*(SRS_LIBS_08525)*

[SWS_Crc_00052] ⌈The `Crc_CalculateCRC8` function of the CRC module shall provide the following CRC results:

Data bytes (hexadecimal)									CRC
00	00	00	00						59
F2	01	83							37
0F	AA	00	55						79
00	FF	55	11						B8
33	22	55	AA	BB	CC	DD	EE	FF	CB
92	6B	55							8C
FF	FF	FF	FF						74

⌋*(SRS_LIBS_08525)*

7.2.1.2 8-bit 0x2F polynomial CRC Calculation

[SWS_Crc_00042] ⌈The `Crc_CalculateCRC8H2F` function of the CRC module shall implement the CRC8 routine based on the generator polynomial 0x2F:

CRC result width:	8 bits
Polynomial:	2Fh
Initial value:	FFh
Input data reflected:	No
Result data reflected:	No
XOR value:	FFh
Check:	DFh
Magic check:	42h

⌋*(SRS_LIBS_08525)*

[SWS_Crc_00053] ⌈The `Crc_CalculateCRC8H2F` function of the CRC module shall provide the following CRC results:

Data bytes (hexadecimal)									CRC
00	00	00	00						12
F2	01	83							C2
0F	AA	00	55						C6
00	FF	55	11						77
33	22	55	AA	BB	CC	DD	EE	FF	11
92	6B	55							33
FF	FF	FF	FF						6C

⌋*(SRS_LIBS_08525)*

7.2.2 16-bit CRC calculation

7.2.2.1 16-bit CCITT-FALSE CRC16

[SWS_Crc_00002] ⌈The CRC module shall implement the CRC16 routine based on the CCITT-FALSE CRC16 Standard:

Note concerning the standard document [5]:

The computed FCS is equal to CRC16 XOR FFFFh when the frame is built (first complement of the CCITT-FALSE CRC16).

For the verification, the CRC16 (CCITT-FALSE) is computed on the same data + FCS, and the resulting value is always 1D0Fh.

Note that, if during the verification, the check would have been done on data + CRC16 (i.e. FCS XOR FFFFh) the resulting value would have been 0000h that is the CCITT-FALSE magic check.

CRC result width:	16 bits
Polynomial:	1021h
Initial value:	FFFFh
Input data reflected:	No
Result data reflected:	No
XOR value:	0000h
Check:	29B1h
Magic check:	0000h

⌋(*SRS_LIBS_08525*)

[SWS_Crc_00054] ⌈The `Crc_CalculateCRC16` function of the CRC module shall provide the following CRC results:

Data bytes (hexadecimal)									CRC
00	00	00	00						84C0
F2	01	83							D374
0F	AA	00	55						2023
00	FF	55	11						B8F9
33	22	55	AA	BB	CC	DD	EE	FF	F53F
92	6B	55							0745
FF	FF	FF	FF						1D0F

⌋(*SRS_LIBS_08525*)

7.2.2.2 16-bit 0x8005 polynomial CRC calculation

[SWS_Crc_00067] ⌈The CRC module shall implement the CRC16 based on the CRC-16/ARC Standard:

CRC result width:	16 bits
Polynomial:	8005h
Initial value:	0000h
Input data reflected:	Yes
Result data reflected:	Yes
XOR value:	0000h
Check:	BB3Dh
Magic check:	0000h

⌋(*SRS_LIBS_08525*)

[SWS_Crc_00068] ⌈The `Crc_CalculateCRC16ARC` function of the CRC module shall provide the following CRC results:

Data bytes (hexadecimal)									CRC
00	00	00	00						0000
F2	01	83							C2E1
0F	AA	00	55						0BE3
00	FF	55	11						6CCF
33	22	55	AA	BB	CC	DD	EE	FF	AE98
92	6B	55							E24E
FF	FF	FF	FF						9401

⌋(*SRS_LIBS_08525*)

7.2.3   32-bit CRC calculation

7.2.3.1   32-bit Ethernet CRC Calculation

[SWS_Crc_00003] ⌈The CRC module shall implement the CRC32 routine based on the IEEE-802.3 CRC32 Ethernet Standard:

CRC result width:	32 bits
Polynomial:	04C11DB7h
Initial value:	FFFFFFFFh
Input data reflected:	Yes
Result data reflected:	Yes
XOR value:	FFFFFFFFh
Check:	CBF43926h
Magic check*:	DEBB20E3h

⌋(*SRS_LIBS_08525*)

*Important note: To match the magic check value, the CRC must be appended in little endian format, i.e. low significant byte first. This is due to the reflections of the input and the result.

[SWS_Crc_00055] ⌈The `Crc_CalculateCRC32` function of the CRC module shall provide the following CRC results:

Data bytes (hexadecimal)									CRC
00	00	00	00						2144DF1C
F2	01	83							24AB9D77
0F	AA	00	55						B6C9B287
00	FF	55	11						32A06212
33	22	55	AA	BB	CC	DD	EE	FF	B0AE863D
92	6B	55							9CDEA29B
FF	FF	FF	FF						FFFFFFFF

⌋(*SRS_LIBS_08525*)

7.2.3.2   32-bit 0xF4ACFB13 polynomial CRC calculation

This 32-bit CRC function is described in [6]. It has an advantage with respect to the Ethernet CRC - it has a Hamming Distance of 6 up to 4kB.

[SWS_Crc_00056] ⌈The CRC module shall implement the CRC32 routine using the 0xF4'AC'FB'13 polynomial:

CRC result width:	32 bits
Polynomial:	F4'AC'FB'13h
Initial value:	FFFFFFFFh
Input data reflected:	Yes
Result data reflected:	Yes
XOR value:	FFFFFFFFh
Check:	16'97'D0'6Ah
Magic check*:	90'4C'DD'BFh
Hamming distance:	6, up to 4096 bytes (including CRC)

⌋*(SRS_LIBS_08525)*

*Important note: To match the magic check value, the CRC must be appended in little endian format, i.e. low significant byte first. This is due to the reflections of the input and the result.

There are three notations for encoding the polynomial, so to clarify, all three notations are shown:

1	Polynomial as binary	0001'1111'0100'1010'1100'1111'1011'0001'0011
2	Normal representation with high bit	01'F4'AC'FB'13h
3	Normal representation	F4'AC'FB'13h
4	Reversed reciprocal representation (=Koopman representation)	FA'56'7D'89h

Notes:

1. Normal representation with high bit = hex representation of polynomial as binary

2. Normal representation with high bit = Koopman representation * 2 + 1

[SWS_Crc_00057] ⌈The `Crc_CalculateCRC32P4` function of the CRC module shall provide the following CRC results:

Data bytes (hexadecimal)									CRC
00	00	00	00						6FB32240h
F2	01	83							4F721A25h
0F	AA	00	55						20662DF8h
00	FF	55	11						9BD7996Eh
33	22	55	AA	BB	CC	DD	EE	FF	A65A343Dh
92	6B	55							EE688A78h
FF	FF	FF	FF						FFFFFFFFh

⌋*(SRS_LIBS_08525)*

7.2.4 64-bit CRC calculation

7.2.4.1 64-bit ECMA polynomial CRC calculation

This 64-bit CRC function is described in [7]. It has a good hamming distance of 4, for long data (see below).

[SWS_Crc_00062] ⌈The CRC module shall implement the CRC64 routine using the 0x42'F0'E1'EB'A9'EA'36'93 polynomial:

CRC result width:	64 bits
Polynomial:	42'F0'E1'EB'A9'EA'36'93h
Initial value:	FFFFFFFFFFFFFFFFh
Input data reflected:	Yes
Result data reflected:	Yes
XOR value:	FFFFFFFFFFFFFFFFh
Check:	99'5D'C9'BB'DF'19'39'FAh
Magic check*:	49'95'8C'9A'BD'7D'35'3Fh
Hamming distance:	4, up to almost 8 GB

⌋*(SRS_LIBS_08525)*

*Important note: To match the magic check value, the CRC must be appended in little endian format, i.e. low significant byte first. This is due to the reflections of the input and the result.

There are three notations for encoding the polynomial, so to clarify, all three notations are shown:

1	Polynomial as binary	0001'0100'0010'1111'0000'1110'0001'1110'1011'1010'1001' 1110'1010'0011'0110'1001'0011
2	Normal representation with high bit	01'42'F0'E1'EB'A9'EA'36'93h
3	Normal representation	42'F0'E1'EB'A9'EA'36'93h
4	Reversed reciprocal representation (=Koopman representation)	A1'78'70'F5'D4'F5'1B'49h

Notes:

1. Normal representation with high bit = hex representation of polynomial as binary

2. Normal representation with high bit = Koopman representation * 2 + 1

[SWS_Crc_00063] ⌈The `Crc_CalculateCRC64` function of the CRC module shall provide the following CRC results:

Data bytes (hexadecimal)									CRC
00	00	00	00						F4A586351E1B9F4Bh
F2	01	83							319C27668164F1C6h
0F	AA	00	55						54C5D0F7667C1575h
00	FF	55	11						A63822BE7E0704E6h
33	22	55	AA	BB	CC	DD	EE	FF	701ECEB219A8E5D5h
92	6B	55							5FAA96A9B59F3E4Eh
FF	FF	FF	FF						FFFFFFFF00000000h

⌋ *(SRS_LIBS_08525)*

7.3 General behavior

Data blocks are passed to the CRC routines using the parameters "start address", "size" and "start value". The return value is the CRC result.

7.4 Version check

For details, refer to the chapter 5.1.8 "Version Check" in SWS_BSWGeneral.

7.5 Debugging concept

None

7.6 Error Classification

Section 7.2 "Error Handling" of the document "General Specification of Basic Software Modules" describes the error handling of the Basic Software in detail. Above all, it constitutes a classification scheme consisting of five error types which may occur in BSW modules.

Based on this foundation, the following section specifies particular errors arranged in the respective subsections below.

7.6.1 Development Errors

There are no development errors.

7.6.2 Runtime Errors

There are no runtime errors.

7.6.3 Transient Faults

There are no transient faults.

7.6.4 Production Errors

There are no production errors.

7.6.5 Extended Production Errors

There are no extended production errors.

8 API specification

8.1 Imported types

In this chapter, all types included from the following modules are listed:

[SWS_Crc_00018] ⌈

Module	Header File	Imported Type
Std	Std_Types.h	Std_VersionInfoType

⌋*()*

8.2 Type definitions

None.

8.3 Function definitions

[SWS_Crc_00013] ⌈If CRC routines are to be used as a library, the CRC modules' implementer shall develop the CRC module in a way that only those parts of the CRC code that are used by other modules are linked into the final binary.⌋*()*

[SWS_Crc_00072] ⌈The CRC library functions shall not call any BSW modules functions (e.g. the DET).⌋*(SRS_LIBS_00018)*

[SWS_Crc_00014] ⌈The CRC function (with parameter *Crc_IsFirstCall* = TRUE) shall do the following operations:

1. As 'Initial value' of the CRC computation, uses the attribute 'Initial value' of the polynomial:

$$Crc = PolynomialInitVal$$

2. If the attribute 'Input data reflected' of the polynomial is TRUE, then reflects input data (byte per byte) obtained via parameters *Crc_DataPtr* and *Crc_Length*:

$$Data = \text{reflect}_8(Data) \text{ (in the case 'Input data reflected' is TRUE)}$$

3. Compute the CRC over the data, the last CRC and the CRC polynomial:

$$Crc = f(Data, Crc, Polynomial)$$

4. Execute the XOR operation between crc and 'XOR value' of the polynomial:

$$Crc = Crc \oplus PolynomialXORVal$$

5. If the attribute 'Result data reflected' of the polynomial is TRUE, then reflect the CRC (over 8, 16 or 32 bits, depending on the CRC size):

$$Crc = reflect_{Crcsize}(Crc) \text{ (in the case 'Result data reflected' is TRUE)}$$

6. The CRC is returned:

$$return\ Crc$$

Steps 2 and 3 are performed as long as data are available⌋*()*

[SWS_Crc_00041] ⌈The CRC function (with parameter *Crc_IsFirstCall* = FALSE) shall do the following operations:

1. As 'Initial value' of the CRC computation, uses the parameter Crc_StartValueX (where X is 8, 8H2F, 16, 32, P4 or 64) that should be the CRC result of the last call. The result is then XORed with 'XOR value' and reflected if 'Result data reflected' of the polynomial is TRUE:

$$Crc = Crc\_StartValueX \oplus PolynomialXORVal$$

$$Crc = \text{reflect}_{Crcsize}(Crc) \text{ (in the case 'Result data reflected' is TRUE)}$$

Steps 2 to 6 are identical to [SWS_Crc_00014].⌋*()*

Usage of CRC functions:

For the first or the unique call the user of a CRC function shall:

1. give a pointer to the data (*Crc_DataPtr*)

2. give the number of bytes of data (*Crc_Length*)

3. give the *Crc_StartValueX* parameter a don't care value (the initialization value is known by the chosen algorithm)

4. give the *Crc_IsFirstCall* parameter the value TRUE to inform the library that it is the first or unique call

5. call the CRC function

6. get the CRC

For the subsequent calls the user has to:

1. give a pointer to the data (*Crc_DataPtr*)

2. give the number of bytes of data (*Crc_Length*)

3. give the *Crc_StartValueX* parameter (X is 8, 8H2F, 16, 32, P4 or 64) the *CRC result of the previous call*

4. give the *Crc_IsFirstCall* parameter the value *FALSE* to inform the library that it is not the first call

5. call the CRC function

6. get the CRC

Example 8.1

Calculation of CRC8: calculation of CRC8 SAEJ1850, over one of test patterns defined by SAE J1850 specification (00h, FFh, 55h, 11h results with CRC B8h)

- If done in one step:

```
1  uint8 Array[4] = {0x00, 0xFF, 0x55, 0x11};
2  uint8 ignored_val = 0x001; /* any value, it is ignored */
3
4  uint8 resultSAE = Crc_CalculateCRC8(&Array[0], 4, ignored_val, TRUE);
```

resultSAE shall be equal to B8h

- If done in several steps:

```
1  uint8 Array[4] = {0x00, 0xFF, 0x55, 0x11};
2  uint8 ignored_val = 0x001; /* any value, it is ignored */
3
4  uint8 resultSAE = Crc_CalculateCRC8(&Array[0], 2, ignored_val, TRUE);
5  resultSAE = Crc_CalculateCRC8(&Array[2], 1, resultSAE, FALSE);
6  resultSAE = Crc_CalculateCRC8(&Array[3], 1, resultSAE, FALSE);
```

resultSAE shall be also equal to B8h

Example 8.2

Calculation of CRC8: calculation of that is not compatible with SAE J1850, but it is compatible with AUTOSAR releases before R4.0:

- If done in one step:

```
1  uint8 Array[4] = {0x00, 0xFF, 0x55, 0x11};
2
3  /* The first call also gets IsFirstCall set to FALSE, and 0xFF as start
        value, which is immediately XORed with 0xFF by CalculateCRC8,
        resulting with start value equal to 0x00. */
4
5  uint8 resultRel3 = Crc_CalculateCRC8(&Array[0], 4, 0xFF, FALSE);
6
7  /* The last XORing must be negated by the caller, to come to 0x00 XOR
        value. */
8
9  resultRel3 = resultR3 ^ 0xFF;
```

resultRel3 contains the same value as computed by AUTOSAR R3.2 CRC8.

- If done in several steps:

```
1  uint8 Array[4] = {0x00, 0xFF, 0x55, 0x11};
2
```

```
3   /* The first call also gets IsFirstCall set to FALSE, and 0xFF as start
        value, which is immediately XORed with 0xFF by CalculateCRC8,
        resulting with start value equal to 0x00. */
4
5   uint8 resultRel3 = Crc_CalculateCRC8(&Array[0], 2, 0xFF, FALSE);
6   resultRel3 = Crc_CalculateCRC8(&Array[2], 1, resultRel3,FALSE);
7   resultRel3 = Crc_CalculateCRC8(&Array[3], 1, resultRel3, FALSE);
8
9   /* The last XORing must be negated by the caller, to come to 0x00 XOR
        value. */
10
11  resultRel3 = resultR3 ^ 0xFF; }
```

resultRel3 contains also the same value as computed by AUTOSAR R3.2 CRC8.

Example 8.3

Calculation of CRC32 Ethernet Standard (see detailed parameters in [SWS_Crc_00003]) over data bytes 01h 02h 03h 04h 05h 06h 07h 08h:

- In one function call, CRC over 01h 02h 03h 04h 05h 06h 07h 08h, start value FFFFFFFFh:

    - CRC-result = **3FCA88C5h** (final value)

- In two function calls:

    - CRC over 01h 02h 03h 04h, start value FFFFFFFFh:

        * CRC-result of first call = B63CFBCDh (intermediate value)

    - CRC over 05h 06h 07h 08h, start value: B63CFBCDh xor *XOR value* (FFFFFFFFh) = 49C30432h and after reflection: 4C20C392h

        * CRC-result of final call = **3FCA88C5h** (final value)

The following C-code example shows that the caller modifies the start value by using the previous result (without any rework) and indicates that it is no more the first call:

```
1   InterResult = Crc_CalculateCRC32(&Array12345678[0], 4, 0xFFFFFFFF, TRUE
        );
2   result = Crc_CalculateCRC32(&Array12345678[4], 4, InterResult, FALSE);
```


8.3.1   8-bit CRC Calculation

8.3.1.1   8-bit SAE J1850 CRC Calculation

[SWS_Crc_00031] ⌈

Service Name	Crc_CalculateCRC8	

Syntax	`uint8 Crc_CalculateCRC8 (`
	`  const uint8* Crc_DataPtr,`
	`  uint32 Crc_Length,`
	`  uint8 Crc_StartValue8,`
	`  boolean Crc_IsFirstCall`
	`)`	

Service ID [hex]	0x01	
Sync/Async	Synchronous	
Reentrancy	Reentrant	
Parameters (in)	Crc_DataPtr	Pointer to start address of data block to be calculated.
	Crc_Length	Length of data block to be calculated in bytes.
	Crc_StartValue8	Start value when the algorithm starts.
	Crc_IsFirstCall	TRUE: First call in a sequence or individual CRC calculation; start from initial value, ignore Crc_StartValue8. FALSE: Subsequent call in a call sequence; Crc_StartValue8 is interpreted to be the return value of the previous function call.
Parameters (inout)	None	
Parameters (out)	None	
Return value	uint8	8 bit result of CRC calculation.
Description	This service makes a CRC8 calculation on Crc_Length data bytes, with SAE J1850 parameters	
Available via	Crc.h	

⌋*(SRS_LIBS_00005, SRS_LIBS_00009, SRS_LIBS_00011, SRS_LIBS_08521)*

[SWS_Crc_00032] ⌈The function `Crc_CalculateCRC8` shall perform a CRC8 calculation using polynomial 0x1D on `Crc_Length` data bytes, pointed to by `Crc_DataPtr`, with the starting value of `Crc_StartValue8`.⌋*(SRS_LIBS_08525)*

[SWS_Crc_00033] ⌈If the CRC calculation within the function `Crc_CalculateCRC8` is performed by hardware, then the CRC module's implementer shall ensure reentrancy of this function by implementing a (software based) locking mechanism.⌋*(SRS_LIBS_-08518, SRS_LIBS_08526)*

Note: If large data blocks have to be calculated (>32 bytes, depending on performance of processor platform), the table based calculation method should be configured for the function `Crc_CalculateCRC8` in order to decrease the calculation time.

The function `Crc_CalculateCRC8` requires specification of configuration parameters defined in `Crc8Mode`.

8.3.1.2 8-bit 0x2F polynomial CRC Calculation

[SWS_Crc_00043] ⌈

Service Name	Crc_CalculateCRC8H2F	

Syntax	`uint8 Crc_CalculateCRC8H2F (`
	`  const uint8* Crc_DataPtr,`
	`  uint32 Crc_Length,`
	`  uint8 Crc_StartValue8H2F,`
	`  boolean Crc_IsFirstCall`
	`)`	

Service ID [hex]	0x05	
Sync/Async	Synchronous	
Reentrancy	Reentrant	
Parameters (in)	Crc_DataPtr	Pointer to start address of data block to be calculated.
	Crc_Length	Length of data block to be calculated in bytes.
	Crc_StartValue8H2F	Start value when the algorithm starts.
	Crc_IsFirstCall	TRUE: First call in a sequence or individual CRC calculation; start from initial value, ignore Crc_StartValue8H2F. FALSE: Subsequent call in a call sequence; Crc_StartValue8H2F is interpreted to be the return value of the previous function call.
Parameters (inout)	None	
Parameters (out)	None	
Return value	uint8	8 bit result of CRC calculation.
Description	This service makes a CRC8 calculation with the Polynomial 0x2F on Crc_Length	
Available via	Crc.h	

⌋*(SRS_LIBS_00005, SRS_LIBS_00009, SRS_LIBS_00011, SRS_LIBS_08521)*

[SWS_Crc_00044] ⌈The function `Crc_CalculateCRC8H2F` shall perform a CRC8 calculation with the polynomial 0x2F on `Crc_Length` data bytes, pointed to by `Crc_DataPtr`, with the starting value of `Crc_StartValue8H2F`.⌋*(SRS_LIBS_08525)*

[SWS_Crc_00045] ⌈If the CRC calculation within the function `Crc_CalculateCRC8H2F` is performed by hardware, then the CRC module's implementer shall ensure reentrancy of this function by implementing a (software based) locking mechanism.⌋ *(SRS_LIBS_08518, SRS_LIBS_08526)*

Note: If large data blocks have to be calculated (>32 bytes, depending on performance of processor platform), the table based calculation method should be configured for the function `Crc_CalculateCRC8H2F` in order to decrease the calculation time.

The function `Crc_CalculateCRC8H2F` requires specification of configuration parameters defined `Crc8H2FMode`.

8.3.2 16-bit CRC Calculation

8.3.2.1 16-bit CCITT-FALSE CRC16

[SWS_Crc_00019] ⌈

Service Name	Crc_CalculateCRC16	

Syntax	`uint16 Crc_CalculateCRC16 (`
	`  const uint8* Crc_DataPtr,`
	`  uint32 Crc_Length,`
	`  uint16 Crc_StartValue16,`
	`  boolean Crc_IsFirstCall`
	`)`	

Service ID [hex]	0x02	
Sync/Async	Synchronous	
Reentrancy	Reentrant	
Parameters (in)	Crc_DataPtr	Pointer to start address of data block to be calculated.
	Crc_Length	Length of data block to be calculated in bytes.
	Crc_StartValue16	Start value when the algorithm starts.
	Crc_IsFirstCall	TRUE: First call in a sequence or individual CRC calculation; start from initial value, ignore Crc_StartValue16. FALSE: Subsequent call in a call sequence; Crc_StartValue16 is interpreted to be the return value of the previous function call.
Parameters (inout)	None	
Parameters (out)	None	
Return value	uint16	16 bit result of CRC calculation.
Description	This service makes a CRC16 calculation on Crc_Length data bytes.	
Available via	Crc.h	

⌋*(SRS_LIBS_00005, SRS_LIBS_00009, SRS_LIBS_00011, SRS_LIBS_08521)*

[SWS_Crc_00015] ⌈The function `Crc_CalculateCRC16` shall perform a CRC16 calculation using polynomial 0x1021 on `Crc_Length` data bytes, pointed to by `Crc_DataPtr`, with the starting value of `Crc_StartValue16`.⌋*(SRS_LIBS_08525)*

[SWS_Crc_00009] ⌈If the CRC calculation within the function `Crc_CalculateCRC16` is performed by hardware, then the CRC module's implementer shall ensure reentrancy of this function by implementing a (software based) locking mechanism.⌋*(SRS_LIBS_08518, SRS_LIBS_08526)*

Note: If large data blocks have to be calculated (>32 bytes, depending on performance of processor platform), the table based calculation method should be configured for the function `Crc_CalculateCRC16` in order to decrease the calculation time.

The function `Crc_CalculateCRC16` requires specification of configuration parameters defined in `Crc16Mode`.

8.3.2.2 16-bit 0x8005 polynomial CRC calculation

[SWS_Crc_00071] ⌈

Service Name	Crc_CalculateCRC16ARC

Syntax	`uint16 Crc_CalculateCRC16ARC (`
	`  const uint8* Crc_DataPtr,`
	`  uint32 Crc_Length,`
	`  uint16 Crc_StartValue16,`
	`  boolean Crc_IsFirstCall`
	`)`

Service ID [hex]	0x08
Sync/Async	Synchronous
Reentrancy	Reentrant
Parameters (in)	Crc_DataPtr	Pointer to start address of data block to be calculated.
	Crc_Length	Length of data block to be calculated in bytes.
	Crc_StartValue16	Start value when the algorithm starts.
	Crc_IsFirstCall	TRUE: First call in a sequence or individual CRC calculation; start from initial value, ignore Crc_StartValue16. FALSE: Subsequent call in a call sequence; Crc_StartValue16 is interpreted to be the return value of the previous function call.
Parameters (inout)	None	
Parameters (out)	None	
Return value	uint16	16 bit result of CRC calculation.
Description	This service makes a CRC16 calculation on Crc_Length data bytes, using the polynomial 0x8005.	
Available via	Crc.h	

⌋*(SRS_LIBS_00005, SRS_LIBS_00009, SRS_LIBS_00011, SRS_LIBS_08521)*

[SWS_Crc_00069] ⌈The function `Crc_CalculateCRC16ARC` shall perform a CRC16 calculation using polynomial 0x8005 on `Crc_Length` data bytes, pointed to by `Crc_-DataPtr`, with the starting value of `Crc_StartValue16`.⌋*(SRS_LIBS_08525)*

[SWS_Crc_00070] ⌈If the CRC calculation within the function `Crc_Calculate-CRC16ARC` is performed by hardware, then the CRC module's implementer shall ensure reentrancy of this function by implementing a (software based) locking mechanism.⌋*(SRS_LIBS_08518, SRS_LIBS_08526)*

Note: If large data blocks have to be calculated (>32 bytes, depending on performance of processor platform), the table based calculation method should be configured for the function `Crc_CalculateCRC16ARC` in order to decrease the calculation time.

The function `Crc_CalculateCRC16ARC` requires specification of configuration parameters defined in `Crc16ARCMode`.

8.3.3 32-bit CRC Calculation

8.3.3.1 32-bit Ethernet CRC Calculation

[SWS_Crc_00020] ⌈

Service Name	Crc_CalculateCRC32	

Syntax	`uint32 Crc_CalculateCRC32 (`
	`  const uint8* Crc_DataPtr,`
	`  uint32 Crc_Length,`
	`  uint32 Crc_StartValue32,`
	`  boolean Crc_IsFirstCall`
	`)`	

Service ID [hex]	0x03	
Sync/Async	Synchronous	
Reentrancy	Reentrant	
Parameters (in)	Crc_DataPtr	Pointer to start address of data block to be calculated.
	Crc_Length	Length of data block to be calculated in bytes.
	Crc_StartValue32	Start value when the algorithm starts.
	Crc_IsFirstCall	TRUE: First call in a sequence or individual CRC calculation; start from initial value, ignore Crc_StartValue32. FALSE: Subsequent call in a call sequence; Crc_StartValue32 is interpreted to be the return value of the previous function call.
Parameters (inout)	None	
Parameters (out)	None	
Return value	uint32	32 bit result of CRC calculation.
Description	This service makes a CRC32 calculation on Crc_Length data bytes.	
Available via	Crc.h	

⌋*(SRS_LIBS_00005, SRS_LIBS_00009, SRS_LIBS_00011, SRS_LIBS_08521)*

[SWS_Crc_00016] ⌈The function `Crc_CalculateCRC32` shall perform a CRC32 calculation using polynomial 0x04C11DB7 on `Crc_Length` data bytes, pointed to by `Crc_DataPtr`, with the starting value of `Crc_StartValue32`.⌋*(SRS_LIBS_08525)*

[SWS_Crc_00010] ⌈If the CRC calculation within the function `Crc_CalculateCRC32` is performed by hardware, then the CRC module's implementer shall ensure reentrancy of this function by implementing a (software based) locking mechanism.⌋*(SRS_LIBS_-08518, SRS_LIBS_08526)*

Note: If large data blocks have to be calculated (>32 bytes, depending on performance of processor platform), the table based calculation method should be configured for the function `Crc_CalculateCRC32` in order to decrease the calculation time.

The function `Crc_CalculateCRC32` requires specification of configuration parameters defined in `Crc32Mode`.

8.3.3.2 32-bit 0xF4ACFB13 polynomial CRC calculation

[SWS_Crc_00058] ⌈

Service Name	Crc_CalculateCRC32P4	

Syntax	```uint32 Crc_CalculateCRC32P4 (
	  const uint8* Crc_DataPtr,
	  uint32 Crc_Length,
	  uint32 Crc_StartValue32,
	  boolean Crc_IsFirstCall
	)```	

Service ID [hex]	0x06	
Sync/Async	Synchronous	
Reentrancy	Reentrant	
Parameters (in)	Crc_DataPtr	Pointer to start address of data block to be calculated.
	Crc_Length	Length of data block to be calculated in bytes.
	Crc_StartValue32	Start value when the algorithm starts.
	Crc_IsFirstCall	TRUE: First call in a sequence or individual CRC calculation; start from initial value, ignore Crc_StartValue32. FALSE: Subsequent call in a call sequence; Crc_StartValue32 is interpreted to be the return value of the previous function call.
Parameters (inout)	None	
Parameters (out)	None	
Return value	uint32	32 bit result of CRC calculation.
Description	This service makes a CRC32 calculation on Crc_Length data bytes, using the polynomial 0x F4ACFB13. This CRC routine is used by E2E Profile 4.	
Available via	Crc.h	

⌋(*SRS_LIBS_00005, SRS_LIBS_00009, SRS_LIBS_00011, SRS_LIBS_08521*)

[SWS_Crc_00059] ⌈The function `Crc_CalculateCRC32P4` shall perform a CRC32 calculation using polynomial 0xF4ACFB13 on `Crc_Length` data bytes, pointed to by `Crc_DataPtr`, with the starting value of `Crc_StartValue32`.⌋(*SRS_LIBS_08525*)

[SWS_Crc_00060] ⌈If the CRC calculation within the function `Crc_Calculate-CRC32P4` is performed by hardware, then the CRC module's implementer shall ensure reentrancy of this function by implementing a (software based) locking mechanism.⌋ (*SRS_LIBS_08518, SRS_LIBS_08526*)

Note: If large data blocks have to be calculated (>32 bytes, depending on performance of processor platform), the (1) hardware supported CRC calculation or (2) table based calculation method, should be configured for the function `Crc_CalculateCRC32P4` in order to decrease the calculation time.

The function `Crc_CalculateCRC32P4` requires specification of configuration parameters defined in `Crc32P4Mode`.

8.3.4 64-bit CRC Calculation

8.3.4.1 64-bit 0x42F0E1EBA9EA3693 polynomial CRC calculation

[SWS_Crc_00061] ⌈

Service Name	Crc_CalculateCRC64	

Syntax	`uint64 Crc_CalculateCRC64 (`
	`  const uint8* Crc_DataPtr,`
	`  uint32 Crc_Length,`
	`  uint64 Crc_StartValue64,`
	`  boolean Crc_IsFirstCall`
	`)`	

Service ID [hex]	0x07	
Sync/Async	Synchronous	
Reentrancy	Reentrant	
Parameters (in)	Crc_DataPtr	Pointer to start address of data block to be calculated.
	Crc_Length	Length of data block to be calculated in bytes.
	Crc_StartValue64	Start value when the algorithm starts.
	Crc_IsFirstCall	TRUE: First call in a sequence or individual CRC calculation; start from initial value, ignore Crc_StartValue64. FALSE: Subsequent call in a call sequence; Crc_StartValue64 is interpreted to be the return value of the previous function call.
Parameters (inout)	None	
Parameters (out)	None	
Return value	uint64	64 bit result of CRC calculation.
Description	This service makes a CRC64 calculation on Crc_Length data bytes, using the polynomial 0x42F0E1EBA9EA3693. This CRC routine is used by E2E Profile 7.	
Available via	Crc.h	

⌋*(SRS_LIBS_00005, SRS_LIBS_00009, SRS_LIBS_00011, SRS_LIBS_08521)*

[SWS_Crc_00064] ⌈The function `Crc_CalculateCRC64` shall perform a CRC64 calculation using polynomial 0x42F0E1EBA9EA3693 on `Crc_Length` data bytes, pointed to by `Crc_DataPtr`, with the starting value of `Crc_StartValue64`.⌋*(SRS_-LIBS_08525)*

[SWS_Crc_00065] ⌈If the CRC calculation within the function `Crc_CalculateCRC64` is performed by hardware, then the CRC module's implementer shall ensure reentrancy of this function by implementing a (software based) locking mechanism.⌋*(SRS_LIBS_-08518, SRS_LIBS_08526)*

Note: If large data blocks have to be calculated (>64 bytes, depending on performance of processor platform), the (1) hardware supported CRC calculation or (2) table based calculation method, should be configured for the function `Crc_CalculateCRC64` in order to decrease the calculation time.

The function `Crc_CalculateCRC64` requires specification of configuration parameters defined in `Crc64Mode`.

8.3.5 Crc_GetVersionInfo

[SWS_Crc_00021] ⌈

Service Name	Crc_GetVersionInfo	

Syntax	`void Crc_GetVersionInfo (`
	`  Std_VersionInfoType* Versioninfo`
	`)`	

Service ID [hex]	0x04	
Sync/Async	Synchronous	
Reentrancy	Reentrant	
Parameters (in)	None	
Parameters (inout)	None	
Parameters (out)	Versioninfo	Pointer to where to store the version information of this module.
Return value	None	
Description	This service returns the version information of this module.	
Available via	Crc.h	

⌋*(SRS_LIBS_00005, SRS_LIBS_00009, SRS_LIBS_00011)*

[SWS_Crc_00011] ⌈The function `Crc_GetVersionInfo` shall return the version information of the CRC module. The version information includes:

- Module Id

- Vendor Id

- Vendor specific version numbers (SRS_BSW_00407).

⌋*(SRS_BSW_00407, SRS_BSW_00411)*

[SWS_Crc_00017] ⌈If source code for caller and callee of the function `Crc_GetVersionInfo` is available, the CRC module should realize this function as a macro, defined in the modules header file.⌋*(SRS_BSW_00407, SRS_BSW_00411)*

8.4 Callback notifications

None.

8.5 Scheduled functions

The Crc module does not have scheduled functions.

8.6 Expected interfaces

In this chapter, all interfaces required from other modules are listed.

8.6.1 Mandatory interfaces

None

8.6.2 Optional interfaces

None.

8.6.3 Configurable interfaces

None.

8.7 Service Interfaces

None.

9 Sequence diagrams

9.1 Crc_CalculateCRC8()

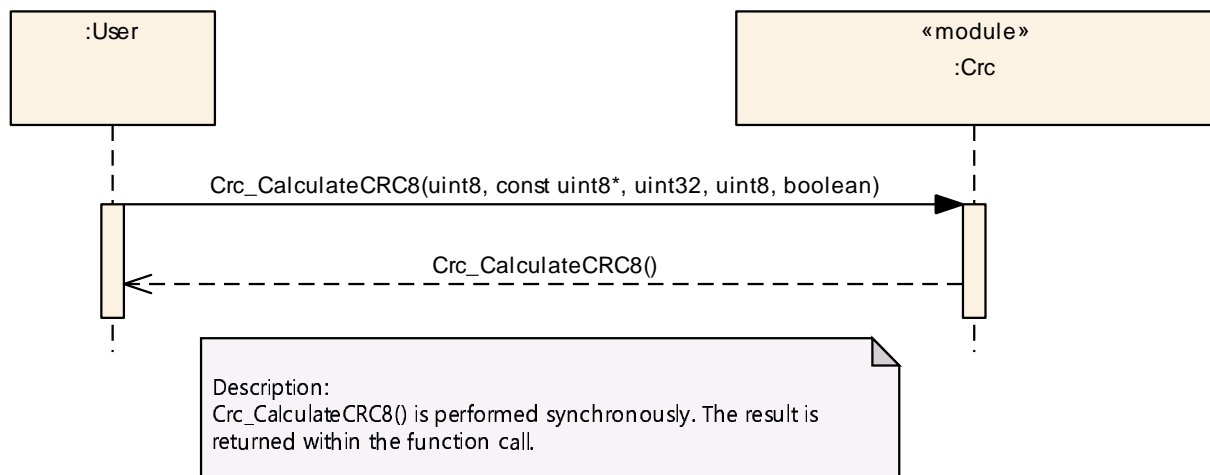The following diagram shows the synchronous function call `Crc_CalculateCRC8`.

Figure 9.1: Crc_CalculateCRC8

9.2 Crc_CalculateCRC8H2F()

The following diagram shows the synchronous function call `Crc_Calculate-CRC8H2F`.
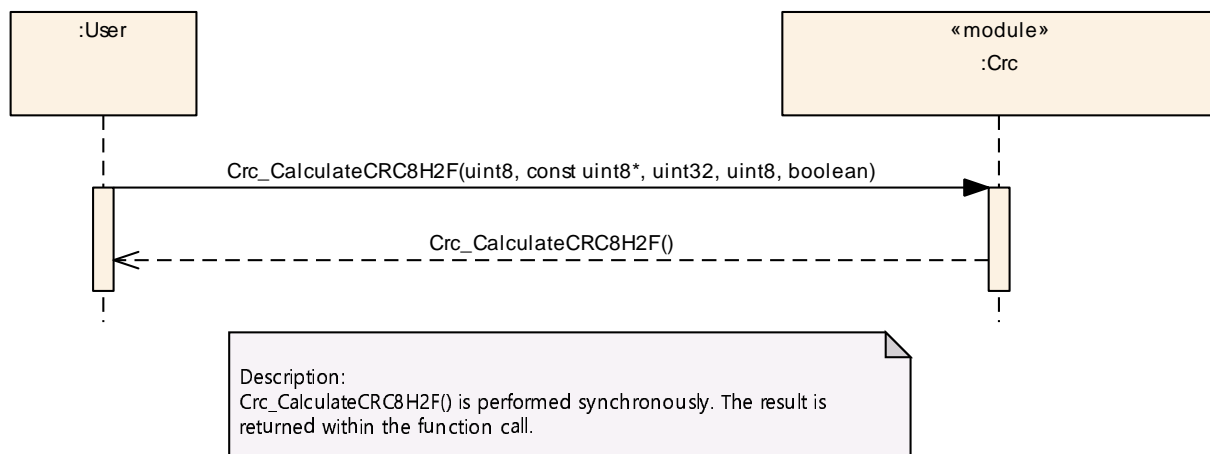

Figure 9.2: Crc_CalculateCRC8H2F

9.3 Crc_CalculateCRC16()

The following diagram shows the synchronous function call Crc_CalculateCRC16.

Figure 9.3: Crc_CalculateCRC16

9.4 Crc_CalculateCRC16ARC()

The following diagram shows the synchronous function call Crc_Calculate-CRC16ARC.

Figure 9.4: Crc_CalculateCRC16ARC

9.5 Crc_CalculateCRC32()

The following diagram shows the synchronous function call Crc_CalculateCRC32.

Figure 9.5: Crc_CalculateCRC32

9.6 Crc_CalculateCRC32P4()

The following diagram shows the synchronous function call Crc_Calculate-CRC32P4.
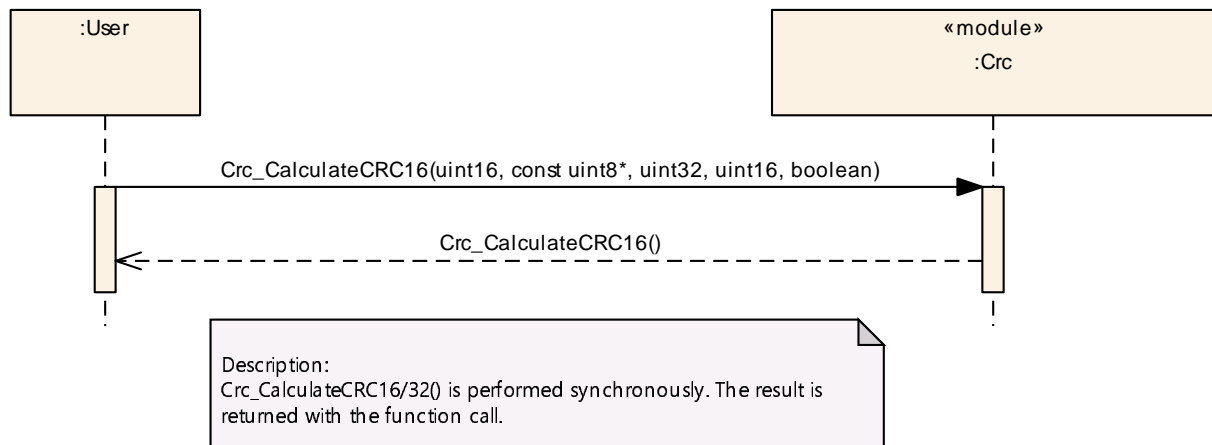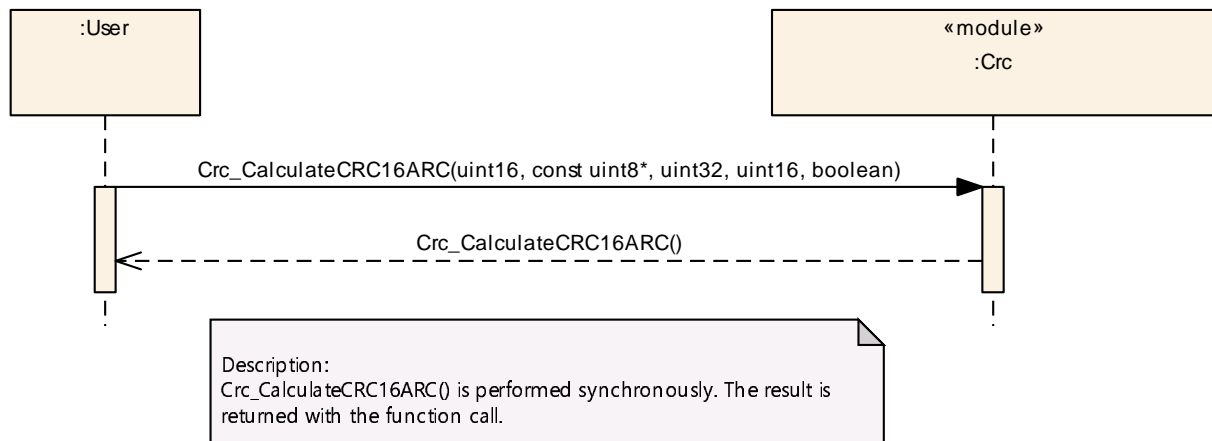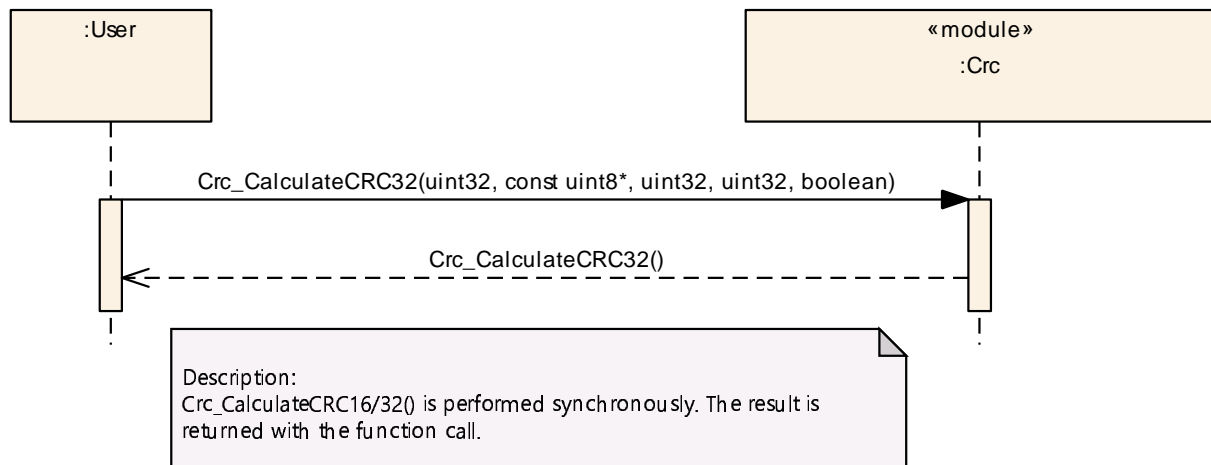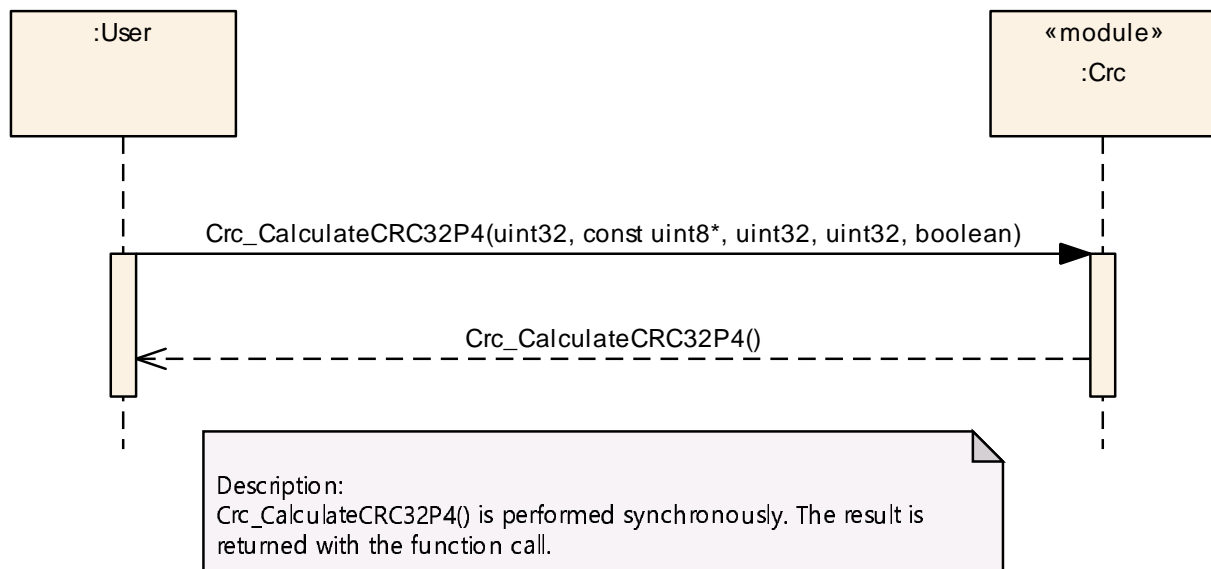

Figure 9.6: Crc_CalculateCRC32P4

9.7 Crc_CalculateCRC64()

The following diagram shows the synchronous function call Crc_CalculateCRC64.
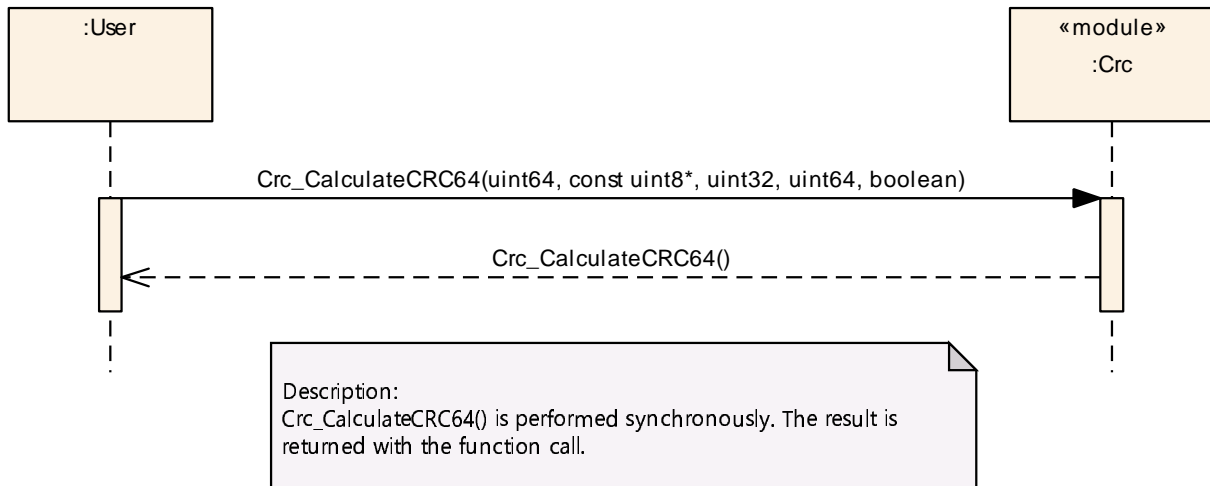

Figure 9.7: Crc_CalculateCRC64

10 Configuration specification

In general, this chapter defines configuration parameters and their clustering into containers. In order to support the specification Chapter 10.1 describes fundamentals. It also specifies a template (table) you shall use for the parameter specification. We intend to leave Chapter 10.1 in the specification to guarantee comprehension.

Chapter 10.2 specifies the structure (containers) and the parameters of the module CRC.

Chapter 10.3 specifies published information of the module CRC.

10.1 How to read this chapter

For details refer to the chapter 10.1 "Introduction to configuration specification" in SWS_BSWGeneral.

In addition to this section, it is highly recommended to read the documents:

- AUTOSAR Layered Architecture [8]

- AUTOSAR ECU Configuration Specification [9]:
  This document describes the AUTOSAR configuration methodology and the AUTOSAR configuration metamodel in detail.

The following is only a short survey of the topic and it will not replace the ECU Configuration Specification document.

10.1.1 Configuration and configuration parameters

Configuration parameters define the variability of the generic part(s) of an implementation of a module. This means that only generic or configurable module implementation can be adapted to the environment (software/hardware) in use during system and/or ECU configuration.

The configuration of parameters can be achieved at different times during the software process: before compile time, before link time or after build time. In the following, the term "configuration class" (of a parameter) shall be used in order to refer to a specific configuration point in time.

10.1.2 Containers

Containers structure the set of configuration parameters. This means:

- all configuration parameters are kept in containers.

- (sub-) containers can reference (sub-) containers. It is possible to assign a multiplicity to these references. The multiplicity then defines the possible number of instances of the contained parameters.

10.2   Containers and configuration parameters

The following chapters summarize all configuration parameters. The detailed meanings of the parameters are described in chapters 7 and 8.
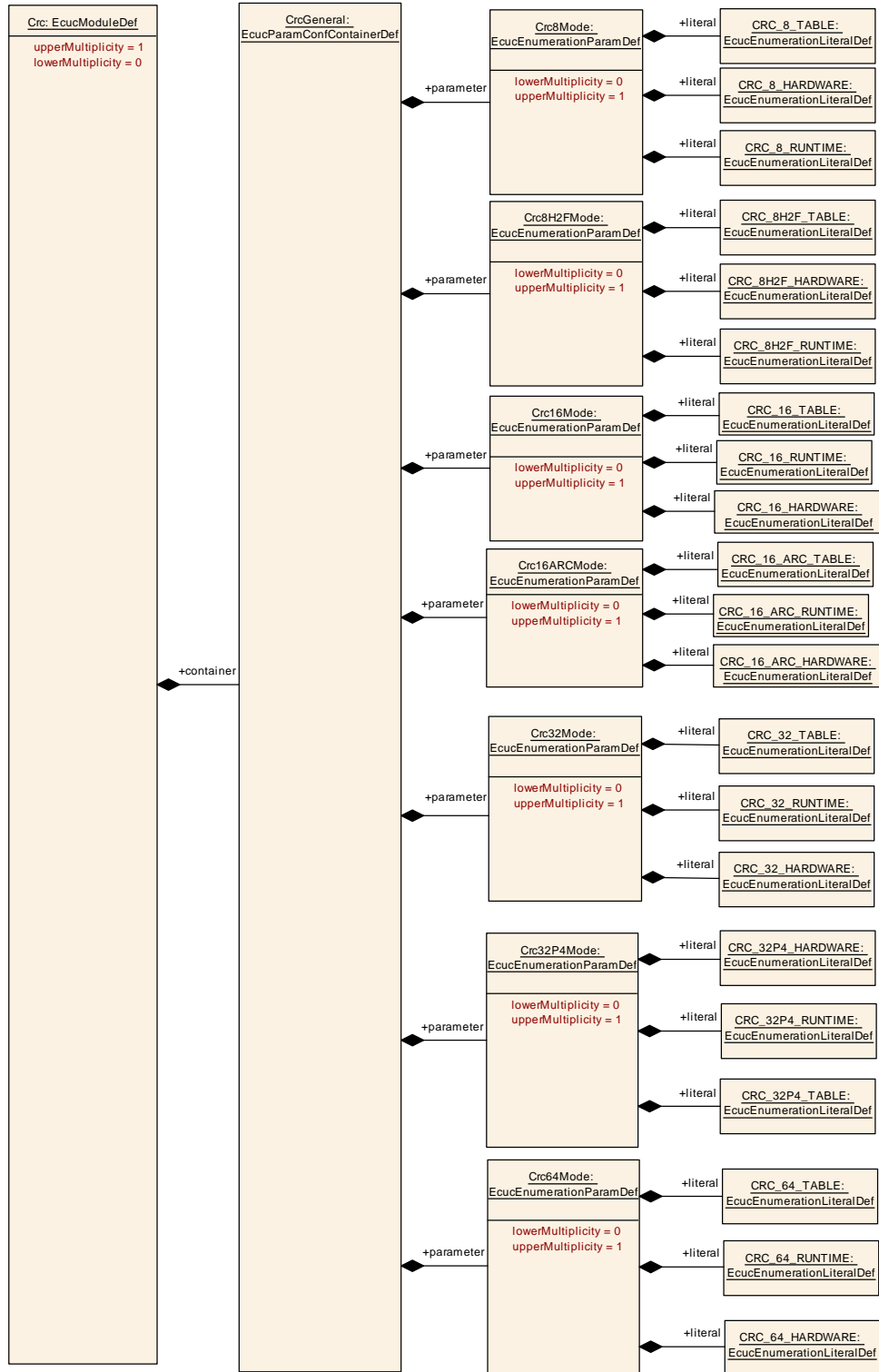

Figure 10.1: CRC

10.2.1 Crc

Module SWS Item	ECUC_Crc_00033		
Module Name	Crc		
Module Description	Configuration of the Crc (Crc routines) module.		
Post-Build Variant Support	false		
Supported Config Variants	VARIANT-PRE-COMPILE		
Included Containers			
Container Name	**Multiplicity**	**Scope / Dependency**	
CrcGeneral	1	General configuration of CRC module	

SWS Item	[ECUC_Crc_00006]
Container Name	CrcGeneral
Parent Container	Crc
Description	General configuration of CRC module
Configuration Parameters	

Name	Crc16ARCMode [ECUC_Crc_00035]		
Parent Container	CrcGeneral		
Description	Switch to select one of the available CRC-16/ARC (polynomial 8005) calculation methods		
Multiplicity	0..1		
Type	EcucEnumerationParamDef		
Range	CRC_16_ARC_HARDWARE	hardware based CRC16 calculation	
	CRC_16_ARC_RUNTIME	runtime based CRC16 calculation	
	CRC_16_ARC_TABLE	table based CRC16 calculation (default selection)	
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: local		

Name	Crc16Mode [ECUC_Crc_00025]		
Parent Container	CrcGeneral		
Description	Switch to select one of the available CRC 16-bit (CCITT) calculation methods		
Multiplicity	0..1		
Type	EcucEnumerationParamDef		
Range	CRC_16_HARDWARE	hardware based CRC16 calculation	
	CRC_16_RUNTIME	runtime based CRC16 calculation	
	CRC_16_TABLE	table based CRC16 calculation (default selection)	
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: local		

Name	Crc32Mode [ECUC_Crc_00026]		
Parent Container	CrcGeneral		
Description	Switch to select one of the available CRC 32-bit (IEEE-802.3 CRC32 Ethernet Standard) calculation methods		
Multiplicity	0..1		
Type	EcucEnumerationParamDef		
Range	CRC_32_HARDWARE	hardware based CRC32 calculation	
	CRC_32_RUNTIME	runtime based CRC32 calculation	
	CRC_32_TABLE	table based CRC32 calculation (default selection)	
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Value Configuration Class	**Pre-compile time**	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: local		

Name	Crc32P4Mode [ECUC_Crc_00032]			
Parent Container	CrcGeneral			
Description	Switch to select one of the available CRC 32-bit E2E Profile 4 calculation methods.			
Multiplicity	0..1			
Type	EcucEnumerationParamDef			
Range	CRC_32P4_HARDWARE	hardware based CRC32P4 calculation		
	CRC_32P4_RUNTIME	runtime based CRC32P4 calculation		
	CRC_32P4_TABLE	table based CRC32P4 calculation (default selection)		
Post-Build Variant Multiplicity	false			
Post-Build Variant Value	false			
Multiplicity Configuration Class	Pre-compile time		X	All Variants
	Link time		–	
	Post-build time		–	
Value Configuration Class	Pre-compile time		X	All Variants
	Link time		–	
	Post-build time		–	
Scope / Dependency	scope: local			

Name	Crc64Mode [ECUC_Crc_00034]			
Parent Container	CrcGeneral			
Description	Switch to select one of the available CRC 64-bit calculation methods.			
Multiplicity	0..1			
Type	EcucEnumerationParamDef			
Range	CRC_64_HARDWARE	hardware based CRC64 calculation		
	CRC_64_RUNTIME	runtime based CRC64 calculation		
	CRC_64_TABLE	table based CRC64 calculation (default selection)		
Post-Build Variant Multiplicity	false			
Post-Build Variant Value	false			
Multiplicity Configuration Class	Pre-compile time		X	All Variants
	Link time		–	
	Post-build time		–	
Value Configuration Class	Pre-compile time		X	All Variants
	Link time		–	
	Post-build time		–	
Scope / Dependency	scope: local			

Name	Crc8H2FMode [ECUC_Crc_00031]		
Parent Container	CrcGeneral		
Description	Switch to select one of the available CRC 8-bit (2Fh polynomial) calculation methods		
Multiplicity	0..1		
Type	EcucEnumerationParamDef		
Range	CRC_8H2F_HARDWARE	hardware based CRC8H2F calculation	
	CRC_8H2F_RUNTIME	runtime based CRC8H2F calculation	
	CRC_8H2F_TABLE	table based CRC8H2F calculation (default selection)	
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	Pre-compile time	X	All Variants
	Link time	–	
	Post-build time	–	
Value Configuration Class	Pre-compile time	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: local		

Name	Crc8Mode [ECUC_Crc_00030]		
Parent Container	CrcGeneral		
Description	Switch to select one of the available CRC 8-bit (SAE J1850) calculation methods		
Multiplicity	0..1		
Type	EcucEnumerationParamDef		
Range	CRC_8_HARDWARE	hardware based CRC8 calculation	
	CRC_8_RUNTIME	runtime based CRC8 calculation	
	CRC_8_TABLE	table based CRC8 calculation (default selection)	
Post-Build Variant Multiplicity	false		
Post-Build Variant Value	false		
Multiplicity Configuration Class	Pre-compile time	X	All Variants
	Link time	–	
	Post-build time	–	
Value Configuration Class	Pre-compile time	X	All Variants
	Link time	–	
	Post-build time	–	
Scope / Dependency	scope: local		

No Included Containers

10.3 Published Information

For details refer to the chapter 10.3 "Published Information" in SWS BSWGeneral [2].

[SWS_Crc_00050] ⌈The standardized common published parameters as required by

SRS_BSW_00402 in the SRS General on Basic Software Modules [3] shall be published within the header file of this module and need to be provided in the BSW ModuleDescription. The according module abbreviation can be found in the List of Basic Software Modules [10].⌋*(SRS_BSW_00402)*

Additional module-specific published parameters are listed below if applicable.

[SWS_Crc_00048] ⌈

Information elements		
Information element name	**Type / Range**	**Information element description**
CRC_VENDOR_ID	#define/ uint16	Vendor ID of the dedicated implementation of this module according to the AUTOSAR vendor list
CRC_MODULE_ID	#define/ uint16	Module ID of this module from Module List
CRC_AR_RELEASE_MAJOR_VERSION	#define/ uint8	Major version number of AUTOSAR release on which the appropriate implementation is based on.
CRC_AR_RELEASE_MINOR_VERSION	#define/ uint8	Minor version number of AUTOSAR release on which the appropriate implementation is based on.
CRC_AR_RELEASE_REVISION_VERSION	#define/ uint8	Patch level version number of AUTOSAR release on which the appropriate implementation is based on.
CRC_SW_MAJOR_VERSION	#define/ uint8	Major version number of the vendor specific implementation of the module. The numbering is vendor specific.
CRC_SW_MINOR_VERSION	#define/ uint8	Minor version number of the vendor specific implementation of the module. The numbering is vendor specific.
CRC_SW_PATCH_VERSION	#define/ uint8	Patch level version number of the vendor specific implementation of the module. The numbering is vendor specific.

⌋*()*

A Not applicable requirements

[SWS_Crc_00051] ⌈These requirements are not applicable to this specification.⌋(
SRS_BSW_00344, *SRS_BSW_00404*, *SRS_BSW_00405*, *SRS_BSW_00170*, *SRS_-
BSW_00383*, *SRS_BSW_00384*, *SRS_BSW_00388*, *SRS_BSW_00389*, *SRS_-
BSW_00395*, *SRS_BSW_00398*, *SRS_BSW_00399*, *SRS_BSW_00400*, *SRS_-
BSW_00401*, *SRS_BSW_00375*, *SRS_BSW_00101*, *SRS_BSW_00416*, *SRS_-
BSW_00406*, *SRS_BSW_00168*, *SRS_BSW_00423*, *SRS_BSW_00424*, *SRS_-
BSW_00425*, *SRS_BSW_00427*, *SRS_BSW_00428*, *SRS_BSW_00429*, *SRS_-
BSW_00432*, *SRS_BSW_00433*, *SRS_BSW_00336*, *SRS_BSW_00337*, *SRS_-
BSW_00369*, *SRS_BSW_00339*, *SRS_BSW_00422*, *SRS_BSW_00417*, *SRS_-
BSW_00323*, *SRS_BSW_00409*, *SRS_BSW_00385*, *SRS_BSW_00386*, *SRS_-
BSW_00161*, *SRS_BSW_00162*, *SRS_BSW_00005*, *SRS_BSW_00415*, *SRS_-
BSW_00164*, *SRS_BSW_00325*, *SRS_BSW_00342*, *SRS_BSW_00343*, *SRS_-
BSW_00160*, *SRS_BSW_00007*, *SRS_BSW_00347*, *SRS_BSW_00305*, *SRS_-
BSW_00307*, *SRS_BSW_00373*, *SRS_BSW_00327*, *SRS_BSW_00335*, *SRS_-
BSW_00350*, *SRS_BSW_00410*, *SRS_BSW_00314*, *SRS_BSW_00348*, *SRS_-
BSW_00353*, *SRS_BSW_00361*, *SRS_BSW_00302*, *SRS_BSW_00328*, *SRS_-
BSW_00312*, *SRS_BSW_00006*, *SRS_BSW_00304*, *SRS_BSW_00378*, *SRS_-
BSW_00306*, *SRS_BSW_00308*, *SRS_BSW_00309*, *SRS_BSW_00358*, *SRS_-
BSW_00414*, *SRS_BSW_00359*, *SRS_BSW_00360*, *SRS_BSW_00330*, *SRS_-
BSW_00331*, *SRS_BSW_00009*, *SRS_BSW_00172*, *SRS_BSW_00010*, *SRS_-
BSW_00333*, *SRS_BSW_00321*, *SRS_BSW_00341*, *SRS_BSW_00334*, *SRS_-
LIBS_00001*, *SRS_LIBS_00002*, *SRS_LIBS_00003*, *SRS_LIBS_00004*, *SRS_LIBS_-
00007*, *SRS_LIBS_00008*, *SRS_LIBS_00010*, *SRS_LIBS_00012*, *SRS_LIBS_00013*,
SRS_LIBS_00015, *SRS_LIBS_00016*, *SRS_LIBS_00017*)