



Elektrobit

EB tresos[®] AutoCore Generic 8 documentation

product release 8.7.1



Elektrobit Automotive GmbH
Am Wolfsmantel 46
91058 Erlangen, Germany
Phone: +49 9131 7701 0
Fax: +49 9131 7701 6333
Email: info.automotive@elektrobit.com

Technical support

<https://www.elektrobit.com/support>

Legal disclaimer

Confidential and proprietary information

ALL RIGHTS RESERVED. No part of this publication may be copied in any form, by photocopy, microfilm, retrieval system, or by any other means now known or hereafter invented without the prior written permission of Elektrobit Automotive GmbH.

All brand names, trademarks and registered trademarks are property of their rightful owners and are used only for description.

Copyright 2019, Elektrobit Automotive GmbH.

Table of Contents

Begin here	9
1. If you are upgrading from a previous release	9
2. If you are using EB tresos AutoCore for the first time	10
3. If you want to start a new project	11
4. If you are looking for how-to instructions	11
5. If you are looking for parameters lists and the API	11
6. If you need help/more information	11
1. EB tresos AutoCore Generic 8 general release notes	13
1.1. Overview	13
1.1.1. Location of up-to-date known problems	13
1.2. Release notes details	15
1.3. Compilers supported	15
1.3.1. Compiler options for GCC 6.2.0	15
1.4. Release notes common to all EB tresos AutoCore Generic 8 modules	16
1.4.1. New features	16
1.4.1.1. Cross-product feature sets with new features	16
1.4.1.2. New features by product	16
1.4.2. Migrating the EB tresos AutoCore Generic modules	18
1.4.3. Limitations and deviations	19
1.4.4. EB-specific enhancements	20
2. About this documentation	21
2.1. Introduction	21
2.2. Required knowledge and system environment	21
2.2.1. Required system environment	21
2.2.2. Required knowledge	22
2.2.3. Required tools	22
2.3. Interpretation of version information	22
2.3.1. Product version number	22
2.3.2. Qualification of basic software	23
2.4. Typography and style conventions	24
2.5. Naming conventions	25
2.5.1. AUTOSAR XML schema	25
2.5.2. Configuration parameter names	26
2.6. EB tresos AutoCore module names	26
3. Safe and correct use of EB tresos AutoCore	31
3.1. Intended usage of EB tresos AutoCore	31
3.2. Possible misuse of EB tresos AutoCore	31
3.3. Target group and required knowledge	31
3.4. Quality standards compliance	31

3.5. Suitability of EB tresos AutoCore for mass production	32
4. Background information	33
4.1. Overview	33
4.2. The AUTOSAR concept	33
4.3. EB tresos AutoCore, the AUTOSAR standard core for automotive ECUs	34
4.3.1. Basic software modules	34
4.3.2. Usage	35
4.3.3. Benefits	35
4.3.4. Features	36
5. Supported features	37
5.1. Overview	37
5.2. Cross-product features	37
5.2.1. Compatibility with different AUTOSAR 4 versions	37
5.2.2. Data transformation and large data communication	38
5.2.3. Diagnostics over IP	38
5.2.4. Global time synchronization	39
5.2.5. Post-build support	40
5.2.5.1. Post-build loadable support	40
5.2.5.2. Post-build selectable support	40
5.2.6. Partial networking	41
5.2.7. BSW distribution	41
5.2.8. Debugging support	42
5.3. Feature extension packages	43
6. Concepts	45
6.1. Overview	45
6.2. Communication stacks	45
6.2.1. Overview	45
6.2.2. Background information	46
6.2.2.1. Communication in the AUTOSAR layered model	46
6.2.2.1.1. PDU concept	47
6.2.2.1.2. Transmission and reception paths	49
6.2.2.1.2.1. Signal-based communication path	49
6.2.2.1.2.2. Diagnostic communication path	50
6.2.2.1.2.3. Signal-based communication I-PDU gateway path	51
6.2.2.1.2.4. Multicast routing path	52
6.2.2.1.3. Handle-ID assignment and PDU linkage	53
6.2.2.1.4. Direct vs. triggered data provision	57
6.2.2.1.4.1. Direct data provision	58
6.2.2.1.4.2. Triggered data provision	60
6.2.2.1.5. Modules and dependencies of the network-dependent communication stack	62
6.3. Network management and state management stack	64

6.3.1. Overview	64
6.3.2. Background information	64
6.3.2.1. Network channels and states	67
6.3.2.2. Network management	68
6.3.2.2.1. Network management algorithm	68
6.3.2.2.2. Network management states	69
6.3.2.2.3. Structure of an NM PDU	72
6.3.2.2.3.1. Control bit vector	73
6.3.2.2.3.2. NM user data	73
6.3.2.3. State management	75
6.3.2.4. Module dependencies	75
6.3.3. Configuring the network and state management stack	76
6.3.3.1. Conditions for configuring the network and state management stack	77
6.3.3.2. Configuring the ComM module	77
6.3.3.3. Configuring the Nm module	79
6.3.3.4. Configuring the bus-dependent network management modules	80
6.3.3.5. Configuring the bus-dependent state management modules	80
6.3.3.6. Integration notes for FrNm	81
6.3.3.7. Integration notes for CanNm	82
6.3.3.8. Integration notes for UdpNm	82
6.4. Adjacent layer property files	82
6.4.1. Overview	82
6.4.2. Background information	82
6.4.2.1. Property format	83
6.4.2.1.1. Property name	83
6.4.2.1.2. Property value	84
6.4.2.2. Property getter XPath functions	84
6.4.2.2.1. Generic signature	84
6.4.2.2.2. Property generalization	85
6.4.2.2.3. Binding free variables	87
6.4.2.3. List of properties and functions	88
6.4.3. Property file	95
6.4.3.1. Property file template	95
6.4.4. Use case: Getting the PDU-ID	100
6.4.4.1. Getting the PDU-ID from own module	100
6.4.4.2. Getting the PDU-ID from adjacent module	101
6.5. Post-build support	102
6.5.1. Overview	102
6.5.2. Background information	102
6.5.2.1. Post-build concept in AUTOSAR	102
6.5.2.1.1. The post-build loadable concept	102
6.5.2.1.2. The post-build selectable concept	103

6.5.2.2. Post-build support in EB tresos AutoCore	104
6.5.2.2.1. Overview of the post-build loadable concept	105
6.5.2.2.2. Post-build selectable concept	107
6.5.2.2.3. Side allocation support	108
6.5.2.2.4. Relocatable configuration data in the post-build concept	108
6.5.2.2.5. Binary code generation in the post-build concept	108
6.5.2.2.6. The post-build configuration manager module <code>PbcfgM</code>	109
6.5.2.2.7. Post-build support in individual modules	111
6.5.2.2.7.1. Understanding the difference between the terms <i>implementation</i> <i>config variant</i> and <i>configuration class</i>	112
6.5.2.2.7.2. Configuration classes of parameters in individual modules	113
6.5.2.2.7.3. Data relocation in individual modules	114
6.5.2.3. Post-build support in EB tresos Studio	114
6.5.2.3.1. Related requirements supported in EB tresos Studio	115
6.5.3. Using post-build support in EB tresos AutoCore	117
6.5.3.1. The post-build configuration workflows	117
6.5.3.2. The workflow for variant handling	118
6.5.3.3. Integration notes	119
6.5.3.3.1. Initializing modules without <code>PbcfgM</code> support	119
6.5.3.3.2. Initializing modules with <code>PbcfgM</code> support	119
6.5.3.3.2.1. Initializing the <code>PbcfgM</code>	120
6.5.3.3.3. Configuration of post-build RAM	120
6.6. AUTOSAR 4.x support	121
6.6.1. Overview	121
6.6.2. Background information	121
6.6.2.1. EB tresos AutoCore Generic 8 in an AUTOSAR 4.x environment	122
6.6.2.2. Overview of the target code	123
6.6.2.3. Interface options for BSW service modules	123
6.6.2.4. EB tresos AutoCore Generic 8 modules that provide 4.x support	124
6.6.3. Configuring the required service APIs	125
6.6.3.1. Use cases	125
6.6.3.2. Configuring steps	125
6.6.4. Generating AUTOSAR 3.2/4.0/4.x schema-compliant BSW SWC descriptions	126
6.7. Measurement and calibration	127
6.7.1. Overview	127
6.7.2. Background information	127
6.7.2.1. Functional overview	128
6.7.2.1.1. Measurement	128
6.7.2.1.2. Calibration	128
6.7.2.1.3. Generation of A2L files	128
6.7.2.2. Modules involved in measurement and calibration	129
6.7.2.3. Modelling measurement and calibration data	131

6.7.3. Using measurement and calibration	131
6.7.3.1. Configuring the Xcp module	132
6.7.3.2. Configuring measurement and calibration data	132
6.8. Multi-core support	133
6.8.1. Overview	133
6.8.2. Background information	134
6.8.2.1. Multi-core approach in AUTOSAR	134
6.8.2.2. Considerations when designing multi-core systems in AUTOSAR	135
6.8.3. Application support	136
6.8.3.1. Communication between cores	137
6.8.3.2. Locking	138
6.8.3.3. Result-free client server communication	138
6.8.4. BSW distribution support	139
6.8.4.1. Concept	139
6.8.4.1.1. Mapping instances of BSW modules to multiple partitions/cores	140
6.8.4.1.2. Distribution of master-satellite BSW modules to multiple partitions/cores	141
6.8.4.2. BSW distribution support in mode management	141
6.8.4.3. BSW distribution of diagnostic log and trace	143
7. Integration	144
7.1. Overview	144
7.2. Integration first steps	144
7.2.1. Recommended task design	144
7.2.1.1. Watchdog task example	145
7.2.2. Software component integration	145
7.2.3. ECU state handling	145
7.3. Integration notes	145
7.3.1. Deviations from MISRA rules	146
7.3.2. Mapping exclusive areas in the basic software modules	146
7.3.2.1. Overview	146
7.3.2.2. Background information	147
7.3.2.3. Configuring the exclusive areas	148
7.3.3. Service Needs Calculator	148
7.3.3.1. Overview	148
7.3.3.2. Background information	149
7.3.3.3. Generic and specific services	150
7.3.3.4. Mainfunction period	151
7.3.3.5. Dem events	151
7.3.3.6. Nvm block	151
7.3.3.7. Init function	152
7.3.3.8. BSW components and their Service Needs	152
7.3.3.9. Functional decomposition	154

7.3.4. Production errors	155
7.3.4.1. Overview	155
7.3.4.2. Background information	155
7.3.4.3. Configuring production errors	155
7.3.5. Memory mapping and compiler abstraction	156
7.3.5.1. Overview	156
7.3.5.2. Background information	156
7.3.5.3. Configuring the memory mapping and compiler abstraction	158
7.3.5.3.1. Generating the memory mapping header files	158
7.3.5.3.2. Editing memory mapping header files	163
7.3.5.3.3. Editing the Compiler_Cfg.h file	164
7.3.5.4. Defining compiler-specific pragma commands	165
7.3.5.4.1. Examples	165
7.4. Optimizing EB tresos AutoCore	167
7.4.1. Overview	167
7.4.2. Background information	167
7.4.2.1. Types of optimization	168
7.4.3. Setting up a module for optimization	168
7.4.4. Optimizing each parameter	170
7.4.5. Optimizing your project as a whole	173
7.4.5.1. Switching off development error detection (Det) checks	173
7.4.5.2. Switching off the <code>GetVersionInfo()</code> API	174
7.4.5.3. Adjusting the exclusive areas configuration	174
8. Bibliography	175

Begin here

1. If you are upgrading from a previous release

If you are upgrading from a previous release, the release notes inform you about the changes. The following release notes documents are provided:

1. `EB tresos AutoCore Generic` general release notes

- ▶ The `EB tresos AutoCore generic` release notes provides the following information that is relevant to all `EB tresos AutoCore Generic` products:
 - ▶ Compiler support and options
 - ▶ Where to find current known problems
 - ▶ New features
 - ▶ Upgrading issues
 - ▶ Limitations and deviations
 - ▶ EB-specific enhancements

The `EB tresos AutoCore generic` release notes are located in [Chapter 1, “EB tresos AutoCore Generic 8 general release notes”](#). To open the release notes in the online help, open `EB tresos Studio` and press **F1**. Navigate to the folder `EB tresos AutoCore Generic` and open the `EB tresos AutoCore Generic 8` documentation.

2. Product-specific release notes

- ▶ The product-specific release notes are contained in the product documentation. These provide the following information that is specific to the modules that are part of the product:
 - ▶ New features
 - ▶ Upgrading issues
 - ▶ Limitations and deviations
 - ▶ EB-specific enhancements

The product documentation is located in the folder `$TRESOS_BASE/doc/4.0_EB_tresos_AutoCore_Generic/`. To open the release notes in the online help, open `EB tresos Studio` and press **F1**. Navigate to the folder `EB tresos AutoCore Generic` and open the product-specific documentation.

3. `MCAL` release notes

- ▶ The `MCAL` release notes provide the scope of delivery of the `MCAL`.

The MCAL release notes are located as PDF at `$TRESOS_BASE/doc/5.0_MCAL_modules/5.1_MCAL_release_notes.pdf` .

To open the release notes in the online help, open EB tresos Studio and press **F1**. Navigate to the folder `MCAL modules` and open the `MCAL release notes`.

MCAL module-specific release information is provided in the respective MCAL guides.

All MCAL guides are located at `$TRESOS_BASE/doc/5.0_MCAL_modules` as PDF file.

In the online help, these guides are located beside the MCAL release notes.

2. If you are using EB tresos AutoCore for the first time

EB tresos AutoCore is the AUTOSAR standard core for automotive ECUs. EB tresos AutoCore's functionality and the AUTOSAR standard behind it are quite complex. If you are a first time user, you may want to get familiar with some of the concepts behind AUTOSAR and its implementation in the EB tresos AutoCore.

► What is EB tresos AutoCore?

The best way to get started is to read through [Chapter 4, “Background information”](#) to familiarize yourself with the AUTOSAR and thus the EB tresos AutoCore concepts.

► Where can I find an example?

The EB tresos AutoCore Generic Base documentation gives you example(s) of EB tresos AutoCore projects along with instructions you may follow for practice.

► How do I configure the modules?

The EB tresos AutoCore Generic product-specific documentation is the place to look when you are ready to actually configure a module or stack:

- In the product-specific documentation, you find a user guide. The user guide gives you detailed information about the concepts and functionalities used within the module or stack in the respective `Background knowledge` chapter.

Following this, the instructional chapter gives you information about configuring the module or stack itself.

- In the product-specific documentation, you find a chapter containing module references. This provides information about the functions, parameters, constants, etc. for each module.

3. If you want to start a new project

The easiest way to create a new project is to start with an example as described in the application examples given in the EB tresos AutoCore Generic Base documentation. The application examples contain all the files you need to start a new project.

4. If you are looking for how-to instructions

All how-to instructions and conceptual information about EB tresos AutoCore can be found in the EB tresos AutoCore Generic product-specific documentation.

5. If you are looking for parameters lists and the API

Parameter lists and the API are described in the module references chapter of the product-specific documentation.

6. If you need help/more information

- ▶ [Chapter 2, “About this documentation”](#)

Find out about:

- ▶ Required knowledge, tools, and system environment
- ▶ Typographical and style conventions used throughout this documentation. Defines usage of special fonts in the documentation.
- ▶ Naming conventions used in this documentation. Defines usage of special names and small/capital lettering in the documentation.
- ▶ Are you looking for a specific document and cannot find it?

Try the top-level EB tresos begin here. As PDF it is located at `$TRESOS_BASE/doc/`. In the online help it is the top-level document EB tresos begin here.

- ▶ Are you looking for a short description of a word or an abbreviation? Refer to the EB tresos glossary. The EB tresos glossary is located at `$TRESOS_BASE/doc/1.0_Installation`.

- ▶ Would you like to read the AUTOSAR specification or other detailed information? Find bibliographic references and further reading suggestions in the bibliography at the end of the product documents.

1. EB tresos AutoCore Generic 8 general release notes

1.1. Overview

Welcome to the EB tresos AutoCore Generic 8 release notes. The release notes are for the target and derivative specified in chapter [Section 1.2, “Release notes details”](#). [Section 1.2, “Release notes details”](#) provides you with information about release versions.

To find out where the MCAL documentation is located, refer to the MCAL release notes.

To find out about known problems (bugs), refer to [Section 1.1.1, “Location of up-to-date known problems”](#).

[Section 1.4, “Release notes common to all EB tresos AutoCore Generic 8 modules”](#) provides information that is common to all modules of EB tresos AutoCore Generic: New features, limitations and deviations, and EB-specific enhancements.

1.1.1. Location of up-to-date known problems

EB continuously updates the list of known problems (also known as *bugs*) so that you are always up-to-date with the information about problems that are inherent in your release. To keep this information as current as possible, known problems are published in a separate PDF document. To download the latest known problems, follow these instructions:

You need internet access and a browser to download the `EB_tresos_Autocore_known_problems.pdf`. You also need access to your EB Command server log-in (alias) and password.


- ▶ Go to https://command.elektrobit.com/command/mod_perl/login.pl.
- ▶ Enter your alias (log-in name) and your password in the respective fields.

If you are not sure, which your alias and your password are, check the email correspondence with EB. You used both alias and password to download the EB tresos AutoCore release you are currently using.


If you have lost alias and password, contact <http://automotive.elektrobit.com/support>.

- ▶ Depending on your account settings, navigate to the project, e.g. `tresos_ASC_V850`. Then navigate to the distribution, e.g. `2009.a_V850_V850ESFx3_RteStandalone`.

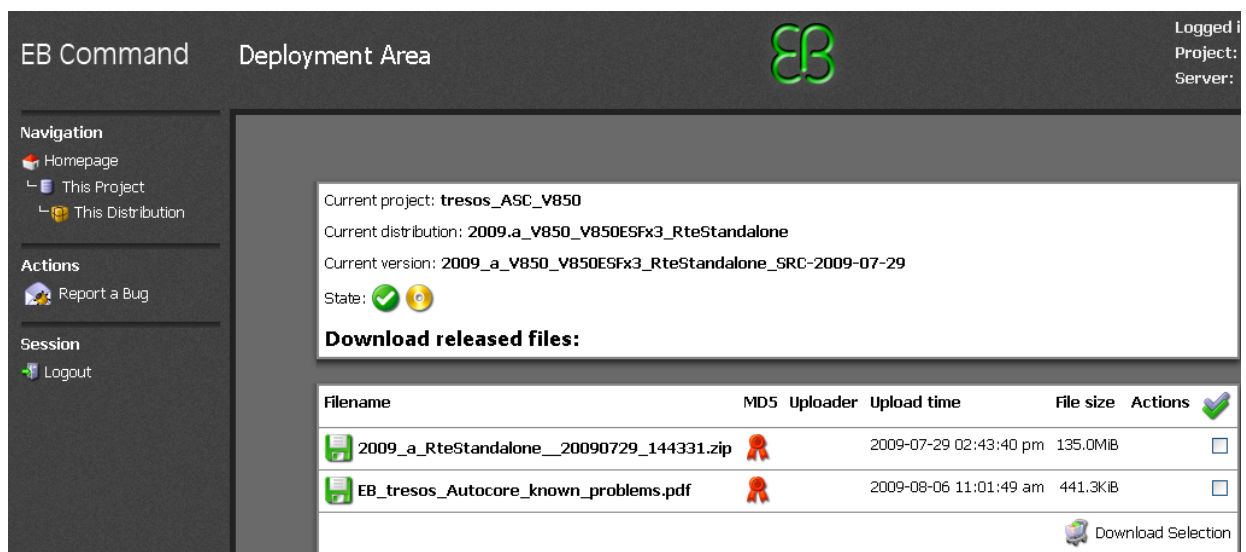



- ▶ Pick the distribution you require by clicking the  button.

The *distribution version* opens up.

- ▶ Pick the distribution version you require by clicking the  button.

The following page opens up:



- ▶ Download `EB_tresos_Autocore_known_problems.pdf` by clicking the **Attachment** button (.

Save the PDF to a location of your choice on your computer.

You are done.

1.2. Release notes details

- ▶ EB tresos AutoCore release version: 8.7.1
- ▶ AUTOSAR R4.0 Rev 3
- ▶ Build number: B271942
- ▶ Target: WINDOWS
- ▶ Derivative: WIN32X86

1.3. Compilers supported

This release of EB tresos AutoCore supports the following compilers:

- ▶ GCC Version 6.2.0

1.3.1. Compiler options for GCC 6.2.0

The compiler options summarize under which conditions the module is to be built. The module is tested using these compiler options. If you change the compiler options, consider the module *untested*.

Compiler	Options
GCC Version 6.2.0	<code>-c -std=iso9899:199409 -ffreestanding -Wpedantic -Wall -Wextra -Wdouble-promotion -Wnull-dereference -Wshift-negative-value -Wshift-overflow -Wswitch-default -Wunused-parameter -Wunused-const-variable=2 -Wuninitialized -Wunknown-pragmas -Wstrict-overflow=2 -Wno-unused-local-typedefs -Warray-bounds=1 -Wduplicated-cond -Wtrampolines -Wfloat-equal -Wdeclaration-after-statement -Wundef -Wno-endif-labels -Wshadow -Wbad-function-cast -Wc90-c99-compat -Wc99-c11-compat -Wcast-qual -Wcast-align -Wwrite-strings -Wdate-time -Wjump-misses-init -Wfloat-conversion -Wno-aggressive-loop-optimizations -Wstrict-prototypes -Wmissing-prototypes -Wmissing-declarations -Wredundant-decls -Wnested-externs -Wvla -Wno-long-long -fno-sanitize-recover -fno-ident -g -O3 -fno-strict-aliasing -fsanitize=undefined -fstack-usage -DNOGDI -D_X86_ -D_WIN32X86_C_GCC_</code>

1.4. Release notes common to all EB tresos AutoCore Generic 8 modules

The following release notes are common to all modules of the EB tresos AutoCore Generic. For more detailed release information concerning the individual modules, see the product-specific documentation.

1.4.1. New features

This release of EB tresos AutoCore Generic 8 contains various improvements. In the following, see the most important ones. For more details on the new features, see the product-specific documentation.

1.4.1.1. Cross-product feature sets with new features

Nothing to report for this release.

1.4.1.2. New features by product

EB tresos AutoCore Generic 8 Base

Feature set	Feature	Feature ID
Software composition editor	Allow the connection of incompatible ports	SWCAS566
Splittable im-/exporter	Allow the filtering of parameters based on the IMPORTER_INFO during export	SPLIMEX53
	Store the IMPORTER_INFO during the import or export of a parameter	SPLIMEX71 SPLIMEX86
	Validate the splittag name for special characters during export	SPLIMEX94

EB tresos AutoCore Generic 8 COM Services

Feature set	Feature	Feature ID
Mode-dependent routing	Support for enabling/disabling a predefined set of PDUs in PduR	COM866
	Multi-frame support of multicast gateway transmission if only one destination PDU is reachable	COM866
ComStack extensions	Support for multicast behavior of PduR based on AUTOSAR 4.1	COM974

EB tresos AutoCore Generic 8 Crypto and Security Stack

Feature set	Feature	Feature ID
Key management	Provide an AUTOSAR-compliant key manager that implements an X.509 certificate parser (new product: EB tresos AutoCore Generic 8 KEYM)	SEC108
Secure onboard communication	Propagate a MAC verification return code to the application	SECOC326

EB tresos AutoCore Generic 8 Diagnostic Stack

Feature set	New feature	Feature ID
Dcm add-ons	Generic end-of-line add-on for DIDs: <ul style="list-style-type: none">▶ Dcm provides an interface to inform CDD about the reception of an unconfigured DID▶ CDD processes DID and informs Dcm about the necessary response	DIAG596

EB tresos AutoCore Generic 8 Diagnostic Stack OBD

Feature set	New feature	Feature ID
Event memory: event combination	Support for combined OBD DTCs	DIAG299

EB tresos AutoCore Generic 8 DLT

Feature set	Feature	Feature ID
Control messages	Support the control message <code>GetSoftwareVersion</code>	DLT414

EB tresos AutoCore Generic 8 FlexRay Stack

Feature set	Feature	Feature ID
Gateway optimization	Avoid sending CF.EOB if upper layer runs out of data	COM848

EB tresos AutoCore Generic 8 IP Stack

Feature set	Feature	Feature ID
Com security	Device authentication (802.1X) supplicant side ^a	COM915

^aFeature is available in prototype quality.

EB tresos AutoCore Generic 8 Memory Stack

Feature set	Feature	Feature ID
Support of writing crash data in Fee	Writing of a high priority block when problems are detected in the system, e.g. OS exception	MEM278
Enhance Fee page swap (erase) handling	Support for enhanced Fee page swap (erase) handling with EB-specific API to cancel a page swap	MEM366

EB tresos AutoCore Generic 8 RTE

Feature set	Feature	Feature ID
Support for mode management	Support mode switch acknowledgement for basic software	RTE162
Support for sender-receiver communication	Support intra-ECU sender-receiver subelement mapping	RTE377

EB tresos AutoCore Generic 8 Watchdog Stack

Feature set	Feature	Feature ID
Multi-core support	Support for executing alive, deadline, and logical monitoring on several cores	WDG388

EB tresos AutoCore Generic 8 XCP

Feature set	Feature	Feature ID
XCP extension	Support for retrieving the identification information by calling a user-defined callout for GET_ID, type 1	XCP425

1.4.2. Migrating the EB tresos AutoCore Generic modules

EB tresos AutoCore Generic provides the functionality to re-use existing module configurations with newer versions of EB tresos AutoCore Generic modules. To convert your existing configurations automatically, specific transformers are provided. For a detailed description on how to use these transformers, refer to [EB tresos Studio user's guide chapter Upgrading projects and configurations](#).

In some cases, the transformers might not be able to convert or configure all configuration parameters for a new version of a module (e.g., if configuration parameters have been added in the new version of the module that cannot be derived from the old configuration parameters). Parameters that are not set by the module transformer will contain default values and therefore may require manual configuration. Parameters containing the default value are indicated in EB tresos Studio with a specific type of icon. Also, to help locate parameters

that use default values, use the EB tresos Studio filtering function as described in the EB tresos Studio user's guide [chapter Editing parameters in the Editor view](#) and [chapter Searching for configuration parameter names and their values](#).

For a description of the changes between different versions of the EB tresos AutoCore Generic modules, please refer to the product-specific documentation. In the module release notes subchapters [Change log](#) a brief description is given for each individual module.

1.4.3. Limitations and deviations

The following limitations and their deviations from the AUTOSAR standard are common to all modules of EB tresos AutoCore Generic.

- ▶ The requirement BSW004 from the SRS General and similar requirements regarding inter-module version checks are not implemented

Description:

The requirement BSW004 from the SRS General and similar requirements regarding inter-module version checks are not implemented. These requirements demand compile-time version checks between the basic software modules.

Rationale:

- ▶ The required compile-time version checks would result in an inflexible, hardly integratable basic software stack.
 - ▶ EB tresos AutoCore is an already integrated product.
 - ▶ The project handling of EB tresos Studio provides means to enforce that only modules with the same AUTOSAR release version can be added to the project.
- ▶ Memory allocation keywords for memory sections with the initialization policies `CLEARED` and `POWER_ON_CLEARED` are not used

Description:

EB tresos AutoCore modules do not use the memory allocation keywords for memory sections with the initialization policies `CLEARED` and `POWER_ON_CLEARED` (e.g., `CANIF_START_SEC_VAR_CLEARED_UNSPECIFIED`).

Rationale:

- ▶ You can achieve a separate mapping of initialized data and zero initialized data to different memory locations with a proper linker command file. Thus, using these explicit memory mappings is superfluous.
- ▶ The `LinNm` module is not delivered with EB tresos AutoCore Generic, which leads to limited functionality for gateway ECUs

Description:

- ▶ EB tresos AutoCore Generic does not provide a `LinNm` module. For non-gateway ECUs, the `LinNm` module has no functionality. Thus, the missing `LinNm` module has no impact on the functionality.
- ▶ To get the `coordinated go to sleep` functionality for gateway ECUs, which the `LinNm` module usually provides, configure your other EB tresos AutoCore Generic modules as follows:
 - ▶ In the `ComM` module, for LIN networks, set `ComMVariant` to `NONE`.
 - ▶ In the `ComM` module, configure an additional `ComMUser` in order to control the LIN network. Map this additional `ComMUser` to the LIN network.
 - ▶ Use the `BswM` module to relay the fact that the CAN network has passed on `COMM_NO_COMMUNICATION` to LIN.
 - ▶ Use the `BswM` to relay the fact that the CAN network has passed on `COMM_FULL_COMMUNICATION` to LIN.
- ▶ The `delayed shutdown via timer` functionality of `LinNm` for gateway ECUs is not supported.

Rationale:

- ▶ `LinNm` only has an impact on gateway-ECUs. For non-gateway ECUs, it does not make a difference whether or not the functionality of the `LinNm` module is provided.
- ▶ Not implementing `LinNm` has only small impact on the functionality, but reduces the code size.
- ▶ Post-build time configuration is supported by selected modules only. (reference to product description: ASCPD-77)

1.4.4. EB-specific enhancements

The following EB-specific enhancements are common to all modules of EB tresos AutoCore Generic.

- ▶ Fulfills selected optimization features according to [HIS-Recommendation for a scalable AUTOSAR Stack](#).

The following generic HIS requirements have been implemented:

- ▶ code size optimization: HisGen0001, HisGen0002, HisGen0003
- ▶ post build enhancements: HisGen0004 (complete FlexRay stack, Com, PduR, IpduM), HisGen0010 (Fr and Com)
- ▶ configurable Dem reporting: HisGen0007, HisGen0008, HisGen0009
- ▶ central memcpy function: HisGen0011, HisGen0012

You find the implemented module-based HIS requirements in the module-specific release notes sections.

2. About this documentation

2.1. Introduction

Welcome to the EB tresos AutoCore Generic 8 documentation.

In this documentation you will find all information necessary to start working with the EB tresos AutoCore Generic.

This chapter `About this documentation` provides you with:

- ▶ Required knowledge, tools, and system environment
- ▶ Typographical and style conventions used throughout this documentation
- ▶ Naming conventions and capitalization rules used in this documentation

NOTE



This documentation may provide information about additional modules

Note that this documentation may contain information, application examples, or instructions about modules that are not part of your order and which you have therefore not received. For example, not all deliveries include the FlexRay communication stack but the chapter `Simple FlexRay demo application example` is always included in the documentation.

2.2. Required knowledge and system environment

2.2.1. Required system environment

- ▶ Microsoft Windows XP/Windows 7 (64-bit)

NOTE



Use of an alternative build environment may lead to non-executable or non-compilable code

Use the build environment delivered with EB tresos AutoCore to execute and compile code. If you use an alternative build environment, your EB tresos AutoCore version may generate non-executable or non-compilable code.

2.2.2. Required knowledge

To work with the EB tresos product line of tools, you need to know the following:

- ▶ The AUTOSAR concept [Section 4.2, “The AUTOSAR concept”](#)
- ▶ How to develop AUTOSAR compliant software components
- ▶ The platform/hardware on which the EB tresos AutoCore should run
- ▶ The compiler and options that can be used

2.2.3. Required tools

To run EB tresos AutoCore you will require the following software tools:

- ▶ EB tresos AutoCore requires EB tresos Studio for configuration and starting the code generation process.

The EB tresos AutoCore is based on the ICC3-compliant AUTOSAR layered architecture and contains modules as listed in [Table 2.1, “Basic software modules and libraries”](#). EB tresos AutoCore consists of the code generators and associated C-code libraries. Essential consistency checks are implemented in code generators for basic software modules.

- ▶ A C compiler and debugger supported by the EB tresos AutoCore. Support of compilers depend upon the target and derivate of EB tresos AutoCore you are using. Please refer to the document EB tresos AutoCore Generic release notes to see which compilers your EB tresos AutoCore supports.
- ▶ Adobe Acrobat Reader to read the pdf documentation.
- ▶ A web browser (e.g. Mozilla Firefox) for contacting support and getting the latest news.

2.3. Interpretation of version information

2.3.1. Product version number

Each product within the product line is assigned an individual *product version number*.

The product version number scheme is made of three parts:

1. The major number

An increment of the *major* version number indicates that the release contains major new features and changes compared to the previous major version.

2. The minor number

An increment of the *minor* version number indicates that the release contains minor new features and changes compared to the previous minor version.

3. The patch number

An increment of the *patch* version number indicates that the release fixes known problems.

The numbers are separated by dots in the following naming scheme: <major>.<minor>.<patch>.

Examples for spelled out product version numbers are:

- ▶ EB tresos AutoCore OS 6.0
- ▶ EB tresos AutoCore OS 6.1
- ▶ EB tresos AutoCore Generic 8.6
- ▶ EB tresos AutoCore Generic 8.7

You can find the product version number for example on the documentation cover and the release notes cover.

Within GUIs, e.g. in EB tresos Studio, you can find the product version number on the splash screen and the **EB tresos details** dialog next to the keyword *Studio*.

There may be a *qualifier* additionally to the product version number. This *qualifier* is provided in brackets and displayed after the product version number.

2.3.2. Qualification of basic software

Basic software of the EB tresos product line are the products: EB tresos AutoCore Generic and EB tresos AutoCore OS.

The quality level of the basic software is provided in the quality statement (Q statement), which is part of each delivery. In the quality statement, the application area is documented based on the quality level of the delivery. The different quality levels and their associated quality criteria are documented in the EB tresos AutoCore Generic Quality Level documentation.

The Quality Level documentation can be found in `$TRESOS_BASE/doc/4.0_EB_tresos_AutoCore_Generic/AutoCore_Generic_Quality_Level_documentation.pdf` where `$TRESOS_BASE` refers to the location of your EB tresos Studio installation.

The quality statement is provided in parallel to the installation file of the basic software and on the command server (EB Command) in the naming scheme: `EB_tresos_AutoCore-quality_statement-<RelName>-<Target>-B<BuildNr>(_<Suffix>).doc`, e.g. `EB_tresos_AutoCore-quality_statement-ACG-6.4-WIN32X86-B64208.doc`

2.4. Typography and style conventions

The signal word *WARNING* indicates information that is vital for the success of the configuration.

WARNING



Source and kind of the problem

What can happen to the software?

What are the consequences of the problem?

How does the user avoid the problem?

The signal word *NOTE* indicates important information on a subject.

NOTE



Important information

Gives important information on a subject

The signal word *TIP* provides helpful hints, tips and shortcuts.

TIP



Helpful hints

Gives helpful hints

Throughout the documentation, you find words and phrases that are displayed in **bold**, *italic*, or monospaced font.

To find out what these conventions mean, see the following table.

All default text is written in Arial Regular font.

Font	Description	Example
Arial italics	Emphasizes new or important terms	The <i>basic building blocks</i> of a configuration are module configurations.
Arial boldface	GUI elements and keyboard keys	1. In the Project drop-down list box, select Project_A.

Font	Description	Example
		2. Press the Enter key.
Monospaced font (Courier)	User input, code, and file directories	The module calls the <code>BswM_Dcm_RequestSessionMode()</code> function. For the project name, enter <code>Project_Test</code> .
Square brackets <code>[]</code>	Denotes optional parameters; for command syntax with optional parameters	<code>insertBefore [<opt>]</code>
Curly brackets <code>{}</code>	Denotes mandatory parameters; for command syntax with mandatory parameters	<code>insertBefore {<file>}</code>
Ellipsis <code>...</code>	Indicates further parameters; for command syntax with multiple parameters	<code>insertBefore [<opt>...]</code>
A vertical bar <code> </code>	Indicates all available parameters; for command syntax in which you select one of the available parameters	<code>allowInvalidMarkup {on off}</code>

2.5. Naming conventions

The naming conventions listed below will enable you to understand meanings of words and word groups that may otherwise be confusing without explanation. This chapter is useful as a reference point if you are unsure what a specific spelling (e.g. capital letters in parameters, a word in boldface that appears on a screen) means. Browse through to see the conventions and return if you need an explanation for a phenomenon you do not understand.

2.5.1. AUTOSAR XML schema

Parameters spelled in capital letters (e.g. `RECOMMENDED-CONFIGURATION-REF` and `BSW-MODULE-DESCRIPTION`) refer to the AUTOSAR XML schema.

XML tags are presented as specified in AUTOSAR Model Persistence Rules for XML V2.1.2 R3.-0 Rev 0001, specifically in chapter 3.6 XML names [\[7\]](#):

- ▶ All XML elements, XML attributes, XML groups and XML types used in the AUTOSAR XML schema are written in upper case letters.
- ▶ In order to increase the readability of the XML names, hyphens are inserted in the XML names which separate the parts of the names.

2.5.2. Configuration parameter names

All configuration parameter names used in EB tresos Studio could be used for code generation and therefore potentially conflict with names used in the EB tresos AutoCore.

WARNING



To avoid naming conflicts, do not use any of the following names as configuration parameter name:

- ▶ any C keywords
- ▶ name of AUTOSAR modules, e.g. Rte, Os, Com.

2.6. EB tresos AutoCore module names

The AUTOSAR specification defines the Basic Software (BSW) module names and library names. The EB implementation of the AUTOSAR BSW is the EB tresos AutoCore; the EB tresos AutoCore uses the same naming convention as AUTOSAR for its modules and libraries.

NOTE



Plug-in and module are synonyms

In this documentation, the terms *module* and *plug-in* are used interchangeably and refer to the modules as defined by AUTOSAR.

- ▶ Plug-in: is the technical `Eclipse` term for the AUTOSAR module to be implemented
- ▶ Module: is the generic term for an AUTOSAR basic software unit as implemented in the EB tresos AutoCore .

Additionally to the AUTOSAR specification, the EB tresos AutoCore uses EB-specific software plug-ins. These plug-ins are located in your EB tresos AutoCore installation in the folder `plugins`. To find out which modules are EB-specific and which ones are affiliated with third parties, see [Table 2.1, “Basic software modules and libraries”](#).

Short name	Module full name ^a	EB product coverage
A2L	A2L	EB tresos AutoCore Generic 8 Base
Adc	ADC Driver	MCAL available from 3rd party providers
ApplDemos	Application Demos	EB tresos AutoCore Generic 8 Base
ApplTemplates	Application Templates	EB tresos AutoCore Generic 8 Base
Base	Base Functionality	EB tresos AutoCore Generic 8 Base
Bfx	Bitfield Functions for Fixed Point Library	Not covered in ACG products, if required contact EB Sales

Short name	Module full name ^a	EB product coverage
BswM	BSW Mode Manager	EB tresos AutoCore Generic 8 Mode Management
BswMAs	BswM Assistant	EB tresos AutoCore Generic 8 Mode Management
Can	CAN Driver for WinCore	EB tresos AutoCore Generic 8 WinCore
CanAs	CAN wizard	EB tresos AutoCore Generic 8 CAN Stack
CanIf	CAN Interface	EB tresos AutoCore Generic 8 CAN Stack
CanNm	CAN Network Management	EB tresos AutoCore Generic 8 CAN Stack
CanSM	CAN State Manager	EB tresos AutoCore Generic 8 CAN Stack
CanTp	CAN Transport Layer	EB tresos AutoCore Generic 8 CAN Stack
CanTrcv	CAN Transceiver Driver	EB tresos AutoCore MCAL 8 CanTrcv
CanTSyn	Time Synchronization over CAN	EB tresos AutoCore Generic 8 Time Sync
Com	Communication	EB tresos AutoCore Generic 8 COM Services
ComM	Communication Manager	EB tresos AutoCore Generic 8 Mode Management
ComXf	COM based Transformer	EB tresos AutoCore Generic 8 Transformers
Configurators	Configurators	EB tresos AutoCore Generic 8 Base
Crc	Cyclic Redundancy Check	EB tresos AutoCore Generic 8 Memory Stack
CryIf	Crypto Interface	EB tresos AutoCore Generic 8 Crypto and Security Stack
Crypto	Crypto Driver	EB tresos AutoCore Generic 8 CRYPTO
Csm	Crypto Service Manager	EB tresos AutoCore Generic 8 Crypto and Security Stack
Dcm	Diagnostic Communication Manager	EB tresos AutoCore Generic 8 Diagnostic Stack
Dem	Diagnostic Event Manager	EB tresos AutoCore Generic 8 Diagnostic Stack
Det	Development Error Tracer	EB tresos AutoCore Generic 8 Base
Dio	Digital IO Driver	MCAL available from 3rd party providers
Dlt	Diagnostic Log and Trace	EB tresos AutoCore Generic 8 DLT (Base)
DoIP	Diagnostic over IP	EB tresos AutoCore Generic 8 DOIP
E2E	End to End Library	EB tresos Safety E2E Wrapper / EB tresos E2E Transformer (E2E)
E2EPW	E2E Protection Wrapper	EB tresos Safety E2E Wrapper
E2EPxx	E2E Profile xx	EB tresos Safety E2E Profile



Short name	Module full name ^a	EB product coverage
E2EXf	E2E Transformer	EB tresos Safety E2E Transformer (E2E)
Ea	EEPROM Abstraction	EB tresos AutoCore Generic 8 Memory Stack
EcuC	ECU Configuration	EB tresos AutoCore Generic 8 Base
EcuM	ECU State Manager	EB tresos AutoCore Generic 8 Mode Management
Eep	EEPROM Driver	EB tresos AutoCore MCAL 8 Eep
Eth	Ethernet Driver	EB tresos AutoCore MCAL 8 Eth
EthIf	Ethernet Interface	EB tresos AutoCore Generic 8 IP Stack
EthSM	Ethernet State Manager	EB tresos AutoCore Generic 8 IP Stack
EthSwt	Ethernet Switch Driver	EB tresos AutoCore MCAL 8 EthSwt
EthTrcv	Ethernet Transceiver Driver	EB tresos AutoCore MCAL 8 EthTrcv
EthTSyn	Time Synchronization over Ethernet	EB tresos AutoCore Generic 8 Time Sync
Fee	Flash EEPROM Emulation	EB tresos AutoCore Generic 8 Memory Stack
FiM	Function Inhibition Manager	EB tresos AutoCore Generic 8 Diagnostic Stack
Fls	Flash Driver for WinCore	EB tresos AutoCore Generic 8 WinCore
FlsTst	Flash Test	MCAL available from 3rd party providers
Fr	FlexRay Driver	EB tresos AutoCore MCAL 8 Fr
FrArTp	FlexRay AUTOSAR Transport Layer	EB tresos AutoCore Generic 8 FlexRay Stack
FrAs	FlexRay Assistant	EB tresos AutoCore Generic 8 FlexRay Stack
FrIf	FlexRay Interface	EB tresos AutoCore Generic 8 FlexRay Stack
FrNm	FlexRay Network Management	EB tresos AutoCore Generic 8 FlexRay Stack
FrSM	FlexRay State Manager	EB tresos AutoCore Generic 8 FlexRay Stack
FrTp	FlexRay ISO Transport Layer	EB tresos AutoCore Generic 8 FlexRay Stack
FrTrcv	FlexRay Transceiver Driver	EB tresos AutoCore MCAL 8 FrTrcv
FrTSyn	Time Synchronization over FlexRay	EB tresos AutoCore Generic 8 Time Sync
Gpt	GPT Driver	MCAL available from 3rd party providers
HidWiz	Calculate Handle IDs	EB tresos AutoCore Generic 8 Base
Icu	ICU Driver	MCAL available from 3rd party providers

Short name	Module full name ^a	EB product coverage
Ifl	Interpolation Floating Point Library	Not covered in ACG products, if required contact EB Sales
Ifx	Interpolation Fixed Point Library	Not covered in ACG products, if required contact EB Sales
IpduM	IPDU Multiplexer	EB tresos AutoCore Generic 8 COM Services
LdCom	Efficient COM for Large Data	EB tresos AutoCore Generic 8 LDCOM
Lin	LIN Driver	MCAL available from 3rd party providers
LinIf	LIN Interface	EB tresos AutoCore Generic 8 LIN Stack
LinNm	LIN Network Management	EB tresos AutoCore Generic 8 LIN Stack
LinSM	LIN State Manager	EB tresos AutoCore Generic 8 LIN Stack
LinTrcv	LIN Transceiver Driver	EB tresos AutoCore MCAL 8 LinTrcv
Make	Application Build Environment	EB tresos AutoCore Generic 8 Base
Mcu	MCU Driver	MCAL available from 3rd party providers
MemAs	Memory Stack Editor	EB tresos AutoCore Generic 8 Memory Stack
MemIf	Memory Abstraction Interface	EB tresos AutoCore Generic 8 Memory Stack
MemMap	Memory Mapping	EB tresos AutoCore Generic 8 Base
Mfl	Mathematical Floating Point Library	Not covered in ACG products, if required contact EB Sales
Mfx	Mathematical Fixed Point Library	Not covered in ACG products, if required contact EB Sales
Nm	Network Management Interface	EB tresos AutoCore Generic 8 Mode Management
NvM	NVRAM Manager	EB tresos AutoCore Generic 8 Memory Stack
Os	Operating System	EB tresos AutoCore OS
PbcfgM	Post-build Configuration Manager	EB tresos AutoCore Generic 8 Base
PduR	PDU Router	EB tresos AutoCore Generic 8 COM Services
Platforms	Platforms	EB tresos AutoCore Generic 8 Base
Port	Port Driver	MCAL available from 3rd party providers
Pwm	PWM Driver	MCAL available from 3rd party providers

Short name	Module full name ^a	EB product coverage
RamTst	RAM Test	EB tresos AutoCore MCAL 8 RamTst
Resource	Resource Plug-in	MCAL available from 3rd party providers
Rte	Runtime Environment	EB tresos AutoCore Generic 8 RTE
SCrc	Safety Cyclic Redundancy Check	EB tresos Safety E2E Transformer (E2E)/EB tresos Safety E2E Wrapper
Sd	Service Discovery	EB tresos AutoCore Generic 8 SD
SecOC	Secure Onboard Communication	EB tresos AutoCore Generic 8 Crypto and Security Stack
SoAd	Socket Adaptor	EB tresos AutoCore Generic 8 IP Stack
SomeIpXf	SOME/IP Transformer	EB tresos AutoCore Generic 8 Transformers
Spi	SPI Handler Driver	MCAL available from 3rd party providers
StbM	Synchronized Time-Base Manager	EB tresos AutoCore Generic 8 Time Sync
SvcAs	Service Needs Calculator	EB tresos AutoCore Generic 8 Base
SwcAs	Software Composition Editor	EB tresos AutoCore Generic 8 Base
TcpIp	TCP/IP	EB tresos AutoCore Generic 8 IP Stack
Time	Time and Execution	EB tresos Safety Time Protection
UdpNm	UDP Network Management	EB tresos AutoCore Generic 8 IP Stack
Wdg	Watchdog Driver	MCAL available from 3rd party providers
WdgIf	Watchdog Interface	EB tresos AutoCore Generic 8 Watchdog Stack
WdgM	Watchdog Manager	EB tresos AutoCore Generic 8 Watchdog Stack
Workflows	Workflows	EB tresos AutoCore Generic 8 Base
Xcp	XCP	EB tresos AutoCore Generic 8 XCP

^aFor an explanation of abbreviations and more details on the modules' functionality, see the glossary which is located in `$TRESOS_BASE/doc/1.0_Installation`.

Table 2.1. Basic software modules and libraries

3. Safe and correct use of EB tresos AutoCore

3.1. Intended usage of EB tresos AutoCore

EB tresos AutoCore is intended to be used in automotive projects based on AUTOSAR (see www.autosar.org).

3.2. Possible misuse of EB tresos AutoCore

- ▶ Use of this product without taking the appropriate risk-reduction measures throughout the entire development phase can result in unexpected behavior. Elektrobit Automotive GmbH is not liable for this misuse. To find out about risk reduction measures, see the **EB tresos maintenance and support annex**. You have received this document together with the product quote supplied by EB.
- ▶ If you use the product in applications that are not defined by the AUTOSAR consortium (see www.autosar.org), the product and its technology may not conform to the requirements of your application. Elektrobit Automotive GmbH is not liable for such misuse.

3.3. Target group and required knowledge

- ▶ Basic software engineers
- ▶ Application developers
- ▶ Programming skills and experience in programming AUTOSAR-compliant ECUs

3.4. Quality standards compliance

EB tresos AutoCore has been developed following processes that have been assessed and awarded Automotive SPICE Level 2. Should it be necessary to include any part of EB tresos AutoCore in a safety relevant project, contact EB first.



3.5. Suitability of EB tresos AutoCore for mass production

The suitability of EB tresos AutoCore for mass production is defined by the mass production review process. Prerequisite for this process is the *quality statement*. This quality statement is available for each release, platform, and derivative of EB tresos AutoCore.

4. Background information

4.1. Overview

The chapter `Background information` is for first-time users of EB tresos AutoCore and those, who would like

- ▶ to update their knowledge of the basic concepts of AUTOSAR and
- ▶ to find out some of the advantages EB tresos AutoCore has to offer.

In each of the following chapters you will also find sections called `Background knowledge`. These provide a conceptual context for the instructions and references that follow in later sections.

While `Background knowledge` chapters may help advanced users of EB tresos AutoCore to figure out why some things are the way they are, they are primarily aimed at newcomers to AUTOSAR and EB tresos AutoCore and those, not very familiar with the graphical user interface.

4.2. The AUTOSAR concept

AUTOSAR (*Automotive Open System Architecture*) is a partnership of

- ▶ automotive,
- ▶ electronics,
- ▶ semiconductor,
- ▶ hard- and software

companies that work toward an open industry standard for automotive embedded architectures.

Software adhering to this industry standard allows you

- ▶ to manage the complicated interaction within automotive Electrical/Electronic (E/E) architectures,
- ▶ to make product upgrades and updates easier and more flexible,
- ▶ to re-use software within a product line and across product lines,
- ▶ to improve E/E systems' quality and reliability,
- ▶ and to find software errors in the early phases of design. Electronic control units (ECU) are the platforms on which the functions of the application (AUTOSAR application software components) are executed.

AUTOSAR functionally separates an ECU's application from the ECU's basic functionalities. An ECU's basic functionalities are

- ▶ the operating system,
- ▶ communication systems,
- ▶ and access to peripherals.

These basic functionalities are implemented in what AUTOSAR calls Basic Software (BSW). All data exchange between application components and the BSW are routed via an abstraction layer. This abstraction layer is the `Run-Time Environment (Rte)`, which hides the BSW layers from the application components. The `Rte` implements the Virtual Function Bus (VFB), which is the abstract communication layer via which the application components actually exchange signals/data. Thus application components exchange data without knowing the path this data is going to take within or between ECUs.

The BSW includes drivers to access peripherals and communication, and the operating system. Application engineers no longer need to or indeed may access the BSW's resources directly but route their call via defined interfaces in the `Rte`. In theory this also enables to move functions from one ECU to another. In practice this is difficult because of signal transmission times and delays.

4.3. EB tresos AutoCore, the AUTOSAR standard core for automotive ECUs

4.3.1. Basic software modules

EB tresos AutoCore is the implementation of AUTOSAR-compliant basic software for automotive ECUs.

The software delivers the infrastructure for running complex control strategies in a multi-bus network environment, including *FlexRay*, *CAN*, and *LIN*. Available for many automotive microcontrollers, the modules are developed in cooperation with semiconductor vendors and tested on target systems to ensure software quality on production level.

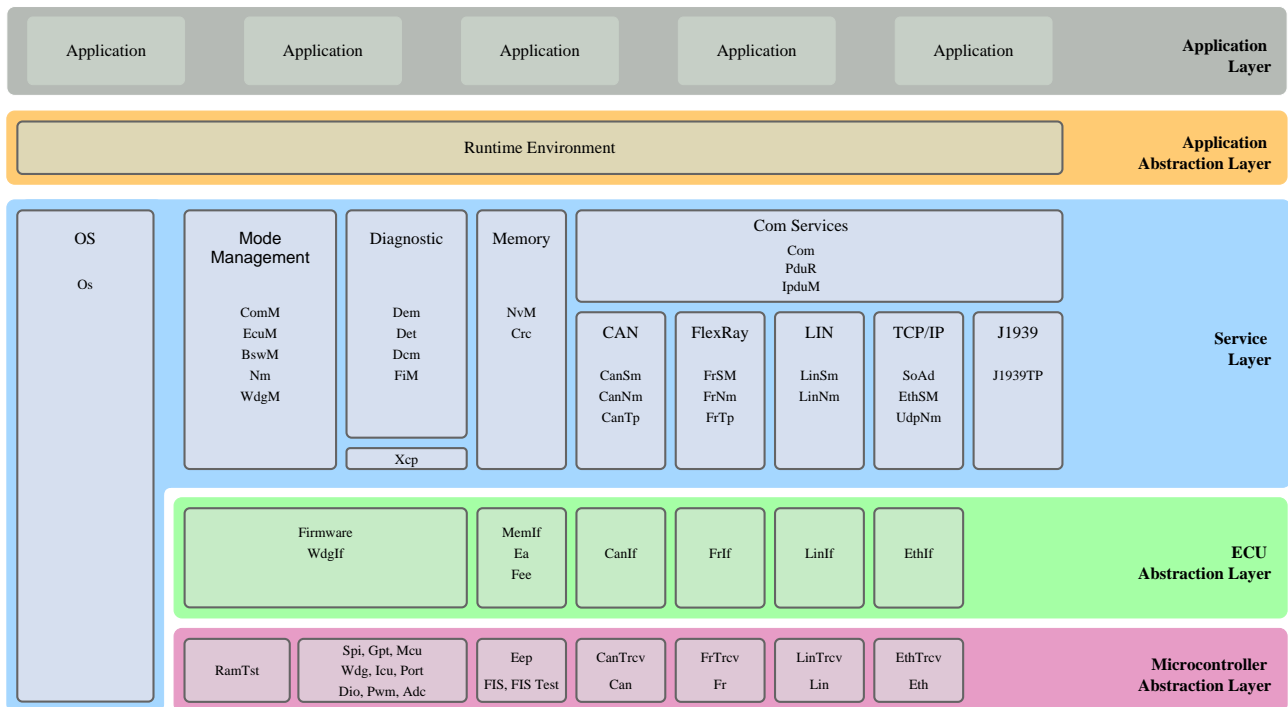


Figure 4.1. The layered architecture of EB tresos AutoCore

4.3.2. Usage

EB tresos AutoCore is well-suited to a wide range of applications, typically multiple supplier projects or projects with a high focus on software re-use, such as:

- ▶ the body, chassis, and powertrain domain,
- ▶ ECUs that require FlexRay, CAN, LIN and gateway functionality,
- ▶ projects that require software in accordance with the AUTOSAR standard.

4.3.3. Benefits

Basic software is needed in any ECU, but excellence in this area is currently not perceived as a competitive advantage. However, companies who buy basic software rather than making it themselves are able to shift critical resources to their core competency, which is applications development.

This is a significant competitive advantage.

The main requirements for an AUTOSAR standard core is its completeness, compliance to the standard and quality on production-level. EB tresos AutoCore meets these requirements.

Besides the economical effects, EB tresos AutoCore has some technical advantages, which further help give you a competitive edge.

4.3.4. Features

EB tresos AutoCore provides:

- ▶ a highly complete AUTOSAR standard core
- ▶ full FlexRay, CAN, and LIN support
- ▶ support for many popular microcontrollers
- ▶ efficient implementation
- ▶ high software quality and maturity features
- ▶ availability of or implementation of all clusters:
 - ▶ operating system (OS),
 - ▶ error handling,
 - ▶ mode management,
 - ▶ memory,
 - ▶ firmware,
 - ▶ COM services,
 - ▶ CAN,
 - ▶ FlexRay,
 - ▶ and LIN.
- ▶ Class-leading implementation of the AUTOSAR run-time environment (RTE) and AUTOSAR OS, with support for scalability classes 1 to 4
- ▶ High software quality due to automotive *SPICE*-compliant development and extensive testing on target hardware
- ▶ Available for major 16- and 32-bit processors from all major hardware vendors, such as Freescale, Fujitsu, Infineon, NEC, Renesas and STMicroelectronics

5. Supported features

5.1. Overview

This chapter contains an overview of features supported in EB tresos AutoCore Generic 8. Features describe a certain functionality provided by the basic software. Cross-product features are provided by more than one EB tresos AutoCore Generic product, e.g. post-build support, whereas product features are provided by one EB tresos AutoCore Generic product only.

This chapter provides information on:

- ▶ cross-product features that are supported since AUTOSAR 4.0.3
- ▶ feature extension packages

For product features, see the product-specific documentation.

5.2. Cross-product features

5.2.1. Compatibility with different AUTOSAR 4 versions

To allow the integration of MCAL or software components developed according to different versions of AUTOSAR 4, some EB tresos AutoCore Generic 8 basic software modules provide compatibility with different AUTOSAR 4 versions as follows:

- ▶ compatibility of the interfaces with MCAL components developed according to different AUTOSAR versions
- ▶ compatibility of the service interfaces for integrating EB tresos AutoCore Generic 8 with software components developed according to different AUTOSAR versions

A configuration parameter allows you to configure the AUTOSAR target version per module. This mechanism is not necessary for modules where the interfaces are compatible between 4.2.1 and 4.0.3. Also the import of system descriptions according to AUTOSAR 4.2.1 is supported.

Affected products:

- ▶ EB tresos Studio for ACG8
- ▶ EB tresos AutoCore Generic 8 Base
- ▶ EB tresos AutoCore Generic 8 CAN Stack

- ▶ EB tresos AutoCore Generic 8 COM Services
- ▶ EB tresos AutoCore Generic 8 Diagnostic Stack
- ▶ EB tresos AutoCore Generic 8 FlexRay Stack
- ▶ EB tresos AutoCore Generic 8 IP Stack
- ▶ EB tresos AutoCore Generic 8 Memory Stack
- ▶ EB tresos AutoCore Generic 8 Mode Management
- ▶ EB tresos AutoCore Generic 8 RTE

5.2.2. Data transformation and large data communication

The EB tresos AutoCore Generic 8 Transformers and Large Data Communication product group facilitates the efficient exchange of large data elements with complex structure for the following use cases:

- ▶ Advanced driver assistance systems (ADAS) over high-bandwidth networks
- ▶ Reduced configuration complexity
- ▶ Deployment of new ECUs that use serialization technology in legacy systems

For further information about the data transformation concept, see the EB tresos AutoCore Generic 8 Transformers documentation.

Affected products:

- ▶ EB tresos AutoCore Generic 8 RTE
- ▶ EB tresos AutoCore Generic 8 Transformers
- ▶ EB tresos AutoCore Generic 8 LDCOM
- ▶ EB tresos Studio for ACG8

5.2.3. Diagnostics over IP

The EB tresos AutoCore Generic 8 DOIP (ACG8 DOIP) product provides the following functions:

- ▶ Diagnostic access to vehicle external testing devices via Ethernet/IP
- ▶ Communication of these external testing devices with diagnostic components inside the vehicle network

The following main features are implemented according to the AUTOSAR 4.1.3 specification:

- ▶ Vehicle network integration
- ▶ Vehicle announcement and vehicle discovery

- ▶ Routing activation
- ▶ Diagnostic message relaying
- ▶ DoIP entity status information

For further information about the Diagnostics over IP concept, see the EB tresos AutoCore Generic 8 DOIP documentation.

Affected products:

- ▶ EB tresos AutoCore Generic 8 DOIP
- ▶ EB tresos AutoCore Generic 8 IP Stack

5.2.4. Global time synchronization

The EB tresos AutoCore Generic 8 Time Sync product group provides a globally agreed notion of time to the other basic software (BSW) modules as well as to the application software components (SWCs) within the complete vehicle network. This agreed notion of time is required to serve the following example use cases:

- ▶ Event data recording
- ▶ Sensor data fusion
- ▶ Vehicle-wide synchronized actions

The following main features are implemented according to the AUTOSAR 4.2.1 specification:

- ▶ Interaction between the StbM module and the network specific >Net<TSyn modules
- ▶ Support for EthTSyn, FrTSyn, and CanTSyn
- ▶ Support for different time-base providers
- ▶ Synchronized execution of runnable entities
- ▶ Provision of current synchronization status
- ▶ Provision of the absolute time value
- ▶ Exchange of synchronization messages

For further information about the global time synchronization concept, see the EB tresos AutoCore Generic 8 Time Sync documentation.

Affected products:

- ▶ EB tresos AutoCore Generic 8 IP Stack
- ▶ EB tresos AutoCore Generic 8 Time Sync
- ▶ EB tresos Studio for ACG8

5.2.5. Post-build support

In line with AUTOSAR, EB tresos AutoCore supports the following two post-build concepts:

- ▶ Post-build loadable
- ▶ Post-build selectable

5.2.5.1. Post-build loadable support

With post-build loadable support, configuration parameter sets can be changed after the ECU basic software has been built and downloaded to the ECU. Therefore, it is not necessary to build the whole ECU software again when only the configuration is changed.

For further information about the post-build loadable concept, see [Section 6.5, “Post-build support”](#).

Affected products:

- ▶ EB tresos AutoCore Generic 8 Base
- ▶ EB tresos AutoCore Generic 8 CAN Stack
- ▶ EB tresos AutoCore Generic 8 COM Services
- ▶ EB tresos AutoCore Generic 8 Crypto and Security Stack
- ▶ EB tresos AutoCore Generic 8 FlexRay Stack
- ▶ EB tresos AutoCore Generic 8 IP Stack
- ▶ EB tresos AutoCore Generic 8 LIN Stack
- ▶ EB tresos AutoCore Generic 8 LDCOM
- ▶ EB tresos AutoCore Generic 8 Mode Management
- ▶ EB tresos ACM 8 CanTrcv
- ▶ EB tresos ACM 8 EthTrcv
- ▶ EB tresos ACM 8 Fr
- ▶ EB tresos Studio for ACG8

5.2.5.2. Post-build selectable support

Post-build selectable support allows you to configure and generate different variants of a configuration, load them into the ECU, and select one of the variants at startup time. This supports the variant handling use case where one ECU binary contains several identities and the ECU decides upon startup which identity to choose.

For further information about the post-build selectable concept, see [Section 6.5, “Post-build support”](#).

Affected products:

- ▶ EB tresos AutoCore Generic 8 Base
- ▶ EB tresos AutoCore Generic 8 CAN Stack
- ▶ EB tresos AutoCore Generic 8 COM Services
- ▶ EB tresos AutoCore Generic 8 Crypto and Security Stack
- ▶ EB tresos AutoCore Generic 8 FlexRay Stack
- ▶ EB tresos AutoCore Generic 8 IP Stack
- ▶ EB tresos AutoCore Generic 8 Mode Management
- ▶ EB tresos ACM 8 Fr
- ▶ EB tresos Studio for ACG8

5.2.6. Partial networking

Partial networking is a mechanism to save power, especially while bus communication is active. Entire ECUs or some functionality within one ECU can be deactivated temporarily if the provided functionality is not required. The partial networking information is exchanged between the corresponding ECUs using network management messages. In order to wake up a deactivated ECU, a special transceiver hardware is required as specified in ISO 11898-5.

Affected products:

- ▶ EB tresos AutoCore Generic 8 Mode Management
- ▶ EB tresos AutoCore Generic 8 CAN Stack
- ▶ EB tresos AutoCore Generic 8 FlexRay Stack
- ▶ EB tresos AutoCore Generic 8 IP Stack
- ▶ EB tresos AutoCore Generic 8 COM Services

5.2.7. BSW distribution

BSW distribution permits the distribution of BSW modules across multiple partitions and cores of an ECU. Any BSW distribution must be separately implemented within each BSW module. A BSW module can be distributed by implementing a master-satellite concept or by instantiation concept. The BSW Scheduler of the ACG8 RTE module provides features for the inter-BSW and intra-BSW module communication.

Affected products:

- ▶ EB tresos AutoCore Generic 8 Base

- ▶ EB tresos AutoCore Generic 8 COM Services
- ▶ EB tresos AutoCore Generic 8 Diagnostic Stack
- ▶ EB tresos AutoCore Generic 8 DLT
- ▶ EB tresos AutoCore Generic 8 Mode Management
- ▶ EB tresos AutoCore Generic 8 RTE Partitioning
- ▶ EB tresos AutoCore Generic 8 Transformers
- ▶ EB tresos AutoCore Generic 8 Watchdog Stack
- ▶ EB tresos Safety E2E Transformer (E2E)

5.2.8. Debugging support

Some BSW modules can provide internal information for debugging with an external debugging tool, e.g. Gliwa T1. Depending on the module's instrumentation with trace macros (*debug hooks*), a module can provide the following information:

- ▶ Function tracing:
 - ▶ Values of the function arguments
 - ▶ Function entry and exit
 - ▶ Return value of the function
- ▶ State tracing:
 - ▶ Old and new state of configured state variables

To find out whether a BSW module is instrumented with trace macros and which functions and states can be traced, check the file `<Mod>_Trace.h`. You only need to implement the macros of interest to you. The unused macros remain empty and therefore do not impact the module performance.



Configuring debugging support

Step 1

In the `Base` module, enable the optional parameter `BaseDbgHeaderFile`.

Step 2

Enter the name of the header file that contains the parameter `BaseDbgHeaderFile`.

Step 3

Ensure that this header file is available in your project.

Step 4

In the header file, implement the macros of the functions/states that you want to trace.

NOTE



Only one header file with BaseDbgHeaderFile

Ensure that you have only one header file that contains the parameter `BaseDbgHeaderFile`. This file is unique and implements all macros from all modules.

5.3. Feature extension packages

Feature extension packages are available for certain releases. A feature extension package contains selected modules with additional features. There are two types of feature extension packages:

- ▶ The features contained take account of the latest feature requests. This type of feature extension package is provided in addition to a regular release to showcase a feature with a lower quality level. So it is possible to already test the functionality.

NOTE



No RFM quality

The modules provided in this feature extension package do not have RFM quality level. Use them for development projects only.

- ▶ The other type of feature extension package contains specific features that are incompatible with the functionality of the regular module release. The incompatible module version has the quality level as defined in the accompanying quality statement. Such a feature extension package stays available for download as long as required.

A feature extension package can be optionally installed in addition to a full ACG release. The modules contained in the package replace the modules of the specific ACG release.

The availability and the content of a feature extension package are announced in the Release Notes and Important Notes provided with the release. The feature extension package has the name of the specific ACG release, extended by *FeatureExtension*. The package can be downloaded as a ZIP file from EB Command.



Downloading and installing a feature extension package

Step 1

Log in to EB Command.

Step 2

Select your ACG release.

Step 3

Download both ZIP files: `ACG-x.x.x.zip` and `ACG-x.x.x_FeatureExtension.zip`.

Step 4

Install the regular release with `ACG-x.x.x.zip`.



Step 5

Install the feature extension with `ACG-x.x.x_FeatureExtension.zip`. This will overwrite some modules.

6. Concepts

6.1. Overview

This chapter describes concepts from AUTOSAR and their implementation in the EB tresos AutoCore Generic product:

- ▶ [Section 6.2, “Communication stacks”](#) describes the concept of communication in AUTOSAR and how this compares to the standard OSI layered communication model.
- ▶ [Section 6.3, “Network management and state management stack ”](#) describes the concept of the network and state management in network-independent and network-dependent modules.
- ▶ [Section 6.4, “Adjacent layer property files”](#) describes the concept of adjacent layer property files.
- ▶ [Section 6.5, “Post-build support”](#) describes the concept of post-build and how this is supported in the EB tresos AutoCore Generic modules.
- ▶ [Section 6.6, “AUTOSAR 4.x support”](#) describes the support for projects that use interfaces from AUTOSAR 4.x.
- ▶ [Section 6.7, “Measurement and calibration”](#) describes the concept of measurement and calibration.
- ▶ [Section 6.8, “Multi-core support”](#) describes the concept of multi-core and how this is supported in the EB tresos AutoCore Generic modules.

6.2. Communication stacks

6.2.1. Overview

This user guide describes the communication stack of EB tresos AutoCore. The communication stack contains a group of modules that simplify communication among ECUs in a vehicle via automotive communication protocols (i.e., CAN, LIN, FlexRay, and Ethernet). [Figure 6.1, “Communication stack overview”](#) illustrates the internal structure of the communication stack.

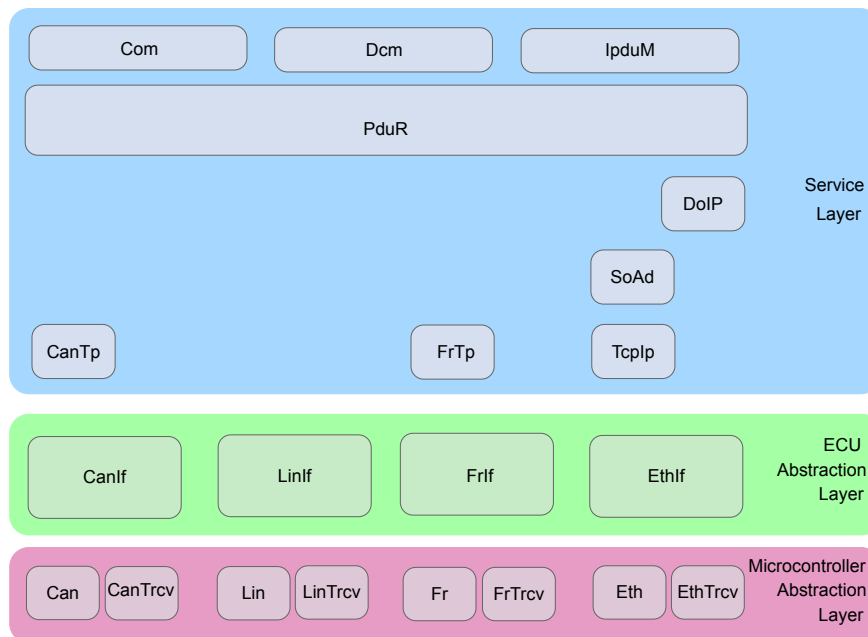


Figure 6.1. Communication stack overview

Logically the communication stack can be divided into a *network-independent* and a *network-dependent* part. While the network-independent part contains only modules that are independent from the used communication protocol (i.e., Com, PduR, Dcm, and IpduM), the network-dependent part depends on the underlying communication protocol (i.e., <Net>, <Net>Trcv, <Net>If, and <Net>Tp). Hereby <Net> acts as a placeholder for the respective prefix for the communication protocol (i.e., Can, Lin, Fr), or Eth .

Refer to the following sections for further details about the communication stack:

- [Section 6.2.2, “Background information”](#) explains the concepts of the communication stack in EB tresos AutoCore.

6.2.2. Background information

This chapter enables you to understand the basic concepts of the AUTOSAR communication stack.

6.2.2.1. Communication in the AUTOSAR layered model

The following sections provide further information about the communication in the AUTOSAR layered model:

- [Section 6.2.2.1.1, “PDU concept”](#) describes the exchange of information entities via the communication stack.

- ▶ [Section 6.2.2.1.2, “Transmission and reception paths”](#) describes the different transmission and reception paths through the communication stack.
- ▶ [Section 6.2.2.1.3, “Handle-ID assignment and PDU linkage”](#) describes the handle-ID assignment and PDU linkage between different modules.
- ▶ [Section 6.2.2.1.4, “Direct vs. triggered data provision”](#) gives an overview of direct and triggered data provisions.

6.2.2.1.1. PDU concept

When describing the information entities exchanged via its communication stack, AUTOSAR uses the terms defined in the ISO standard for *open systems interconnection (OSI)* as defined in [1] and [2]. OSI refers to the different layers in a communication stack via numbers: the layers start with number 1 for the physical layer and end with layer 7 for the application layer. OSI prefixes all entities (e.g., layers, protocol data units (PDUs), ...) of a given layer (N) with this layer number (N) or a one-letter layer identifier (e.g. T for the Transport layer).

The instance of layer (N) of one ECU's communication stack virtually¹ exchanges *protocol data units (PDUs)* with its peer instance of other ECUs.

To accomplish this exchange of (N)-PDUs, the instance of layer (N) uses the services provided by the instance of layer (N-1) on the same ECU. Layer (N) thus calls for API functions of layer (N-1) and passes the (N)-PDU to be transmitted as a parameter to these API functions.

From the perspective of layer (N-1) the (N)-PDU passed by layer (N) is just a chunk of data to be transmitted. OSI uses the term *service data unit ((N-1)-SDU)* for this chunk of data.

On the sending side, layer (N-1) adds a *protocol control information ((N-1)-PCI)* field to this (N-1)-SDU to form the (N-1)-PDU, which is then passed onward to the layer (N-2). This (N-1)-PCI, which contains (N-1)-layer protocol information like e.g., sender and receiver address and checksums, is removed by the instance of layer (N-1) at the receiver side before forwarding the (N-1)-SDU to layer (N). [Figure 6.2, “PDUs, SDUs, and PCIs”](#) illustrates this forming of (N-1)-PDUs out of (N-1)-SDUs and (N-1)-PCIs, as well as the relationship between (N)-PDUs and (N-1)-SDUs.

¹The exchange is just virtual, since the only exchange that *physically* takes place in a data exchange is on the lowest layer (i.e. on layer 1, the physical layer).

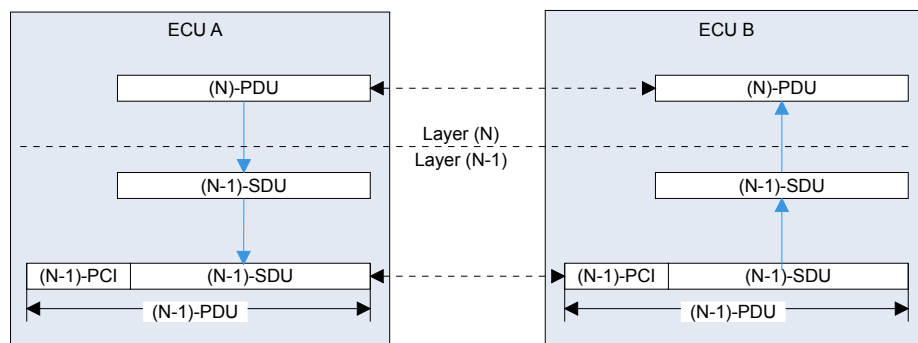


Figure 6.2. PDUs, SDUs, and PCIs

AUTOSAR defines one-letter layer identifiers for the layers relevant to the specification (see [Figure 6.3, “Types of PDUs exchanged in AUTOSAR”](#)):

Data-Link layer (L-layer)

The driver modules (<Net>) and the interface (<Net>If) modules in AUTOSAR are assigned to the Data-Link layer (L-layer). Thus PDUs exchanged between these two modules are termed *L-PDUs*.

Network layer (N-layer)

In contradiction to OSI, the transport protocol modules (<Net>Tp) in AUTOSAR are assigned to the *Network layer (N-layer)*. Thus PDUs passed between the <Net>Tp and the <Net>If modules are called *N-PDUs* in AUTOSAR.

Interaction layer (I-layer)

All modules of the communication stack located between the *Rte* and the <Net>Tp module (or <Net>If module) are assigned to the *Interaction layer (I-layer)*² in AUTOSAR. Thus PDUs exchange between these modules are named *I-PDUs* in AUTOSAR.

²OSI uses the term *Presentation layer*.

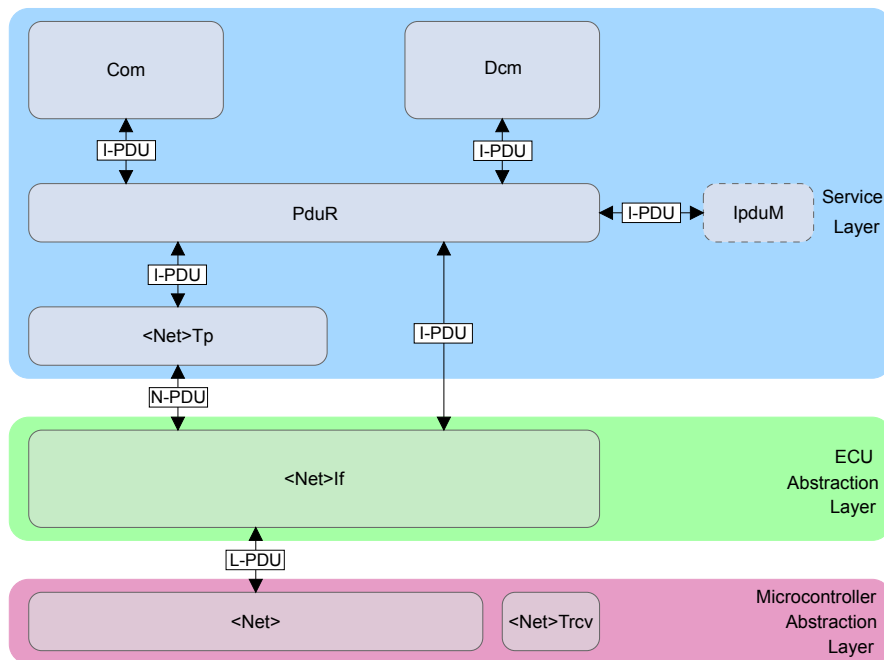


Figure 6.3. Types of PDUs exchanged in AUTOSAR

6.2.2.1.2. Transmission and reception paths

The four different transmission and reception paths through the AUTOSAR communication stack are described in the following sections:

- ▶ [Section 6.2.2.1.2.1, “Signal-based communication path”](#)
- ▶ [Section 6.2.2.1.2.2, “Diagnostic communication path”](#)
- ▶ [Section 6.2.2.1.2.3, “Signal-based communication I-PDU gateway path”](#)
- ▶ [Section 6.2.2.1.2.4, “Multicast routing path”](#)

6.2.2.1.2.1. Signal-based communication path

Signal-based communication uses the driver modules (e.g., **Can**), the interface modules (e.g., **CanIf**), the PDU Router module (**PduR**) and the **Com** module. [Figure 6.4, “Signal-based communication path”](#) illustrates cases, in which an L-PDU (i.e. a frame) is received by the driver module, handed onward to the respective Interface module, and finally routed to **Com** (as I-PDU) by the **PduR**. Upon transmission, the frame uses the same path backwards to the respective driver module.

NOTE



In the FlexRay stack several I-PDUs can be packed into one L-PDU

Note that the FlexRay stack is different than the other stacks, since here multiple I-PDUs can be packed into a single L-PDU when transmitting them and have to be extracted when they are received.

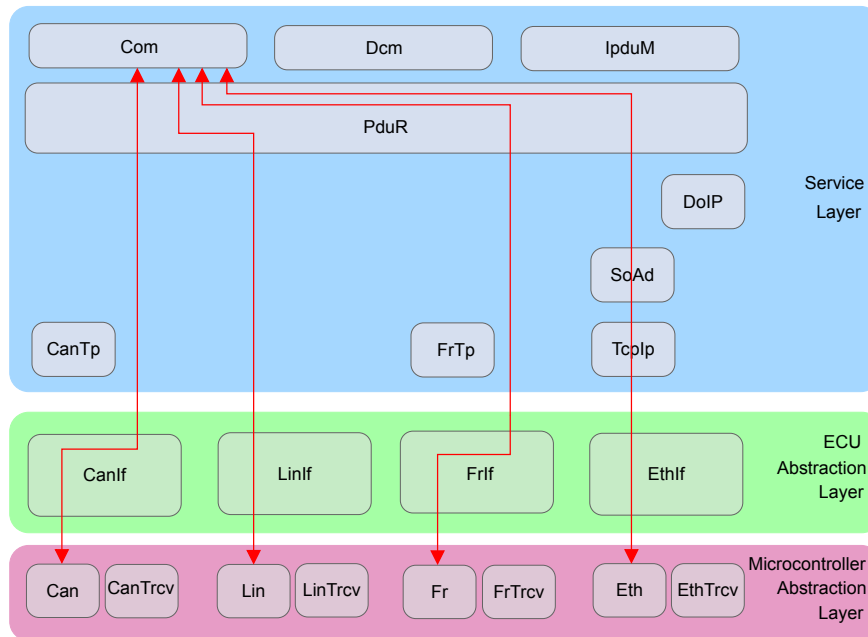


Figure 6.4. Signal-based communication path

6.2.2.1.2.2. Diagnostic communication path

Diagnostic communication uses the modules Driver, Interface, Transport Protocol, PDU Router and *Dcm*. [Figure 6.5, “Diagnostic communication path”](#) illustrates cases, in which multiple L-PDUs (i.e. frames) are received by the Driver module and sent on to the respective Interface module. The Interface module passes these L-PDUs to the Transport Protocol module as N-PDU. The Transport Protocol assembles an I-PDU out of (potentially) multiple N-PDUs. This I-PDU is finally routed to the *Dcm* by the *PduR*. Upon transmission, the frames uses the same path backwards. This requires the Transport Protocol module to segment a (large) I-PDU into multiple N-PDUs .

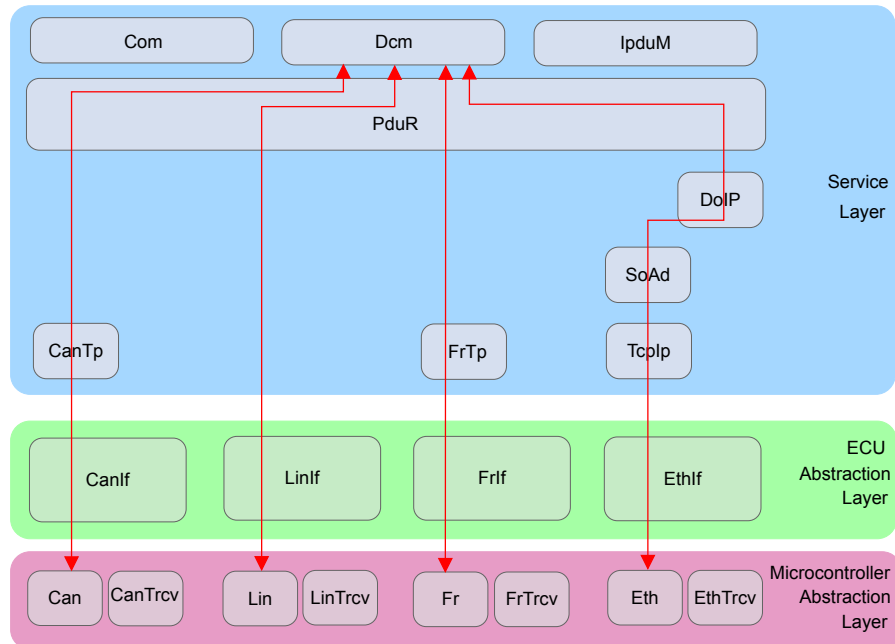


Figure 6.5. Diagnostic communication path

6.2.2.1.2.3. Signal-based communication I-PDU gateway path

The signal-based communication I-PDU gateway uses the modules: driver, interface, and PDU Router. [Figure 6.6, “Signal-based communication I-PDU gateway path”](#) illustrates cases, in which an L-PDU (i.e., a frame) is received by the source driver module, sent on to the respective source interface module and then to the PduR module, which then routes the frame to the destination interface module. From there it is sent to the destination driver module for transmission.

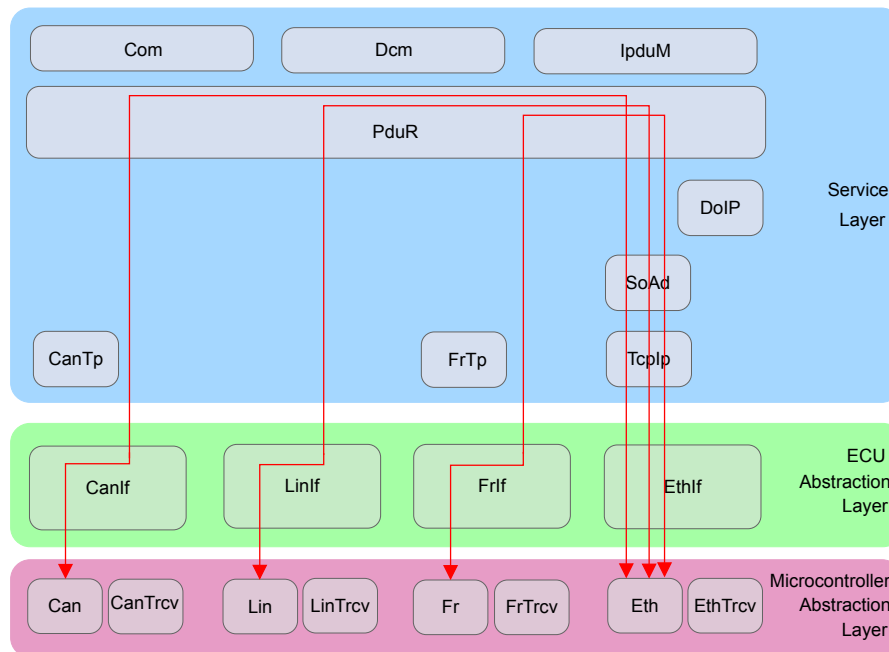


Figure 6.6. Signal-based communication I-PDU gateway path

6.2.2.1.2.4. Multicast routing path

Multicast routing paths support 1:n communication (i.e., one communication source and multiple communication destinations). Basically any of the previously described routing paths can be part of a multicast routing path. In multicast routing paths the `PduR` module is in charge of delivering a single I-PDU to multiple destinations. [Figure 6.7, “Multicast routing path”](#) shows an example, in which an L-PDU (i.e., a frame) is received by the `Can` module of the source network, sent on to the `CanIf` module and routed to the `Com` module for local reception as well as to the `LinIf` and the `FrIf` module for transmission to the respective destination networks.

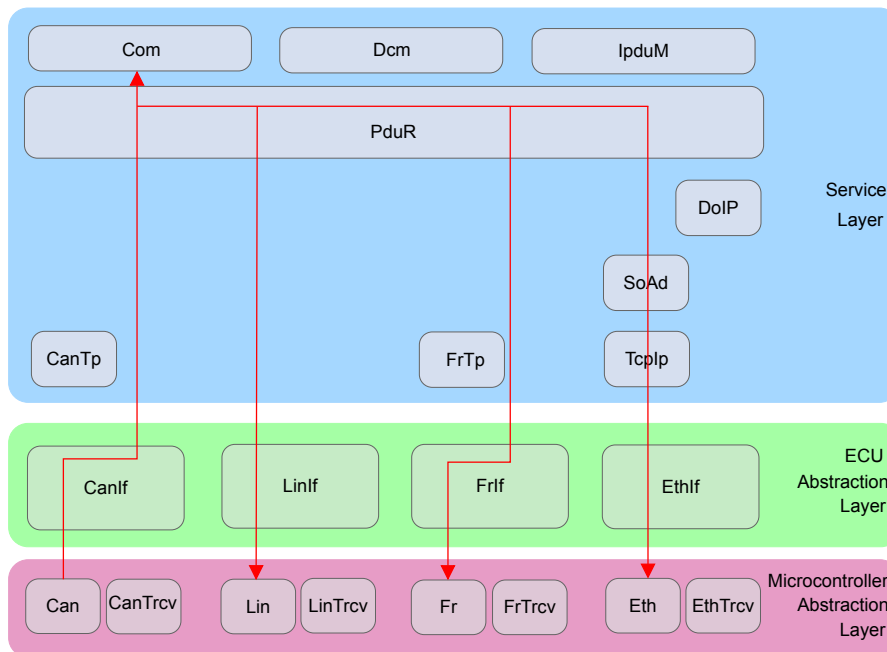


Figure 6.7. Multicast routing path

6.2.2.1.3. Handle-ID assignment and PDU linkage

AUTOSAR defines that most of the API functions of the communication stack obtain two types of parameters:

1. pointers to PDU data buffers, where the data buffer contains the information to be transmitted (i.e, the SDU from the perspective of the lower layer)
2. a *handle-ID* (also called PDU-ID) to identify the PDU

[Figure 6.8, “Typical communication stack API function example”](#) gives an example for a typical communication stack API function:

```
Std_ReturnType PduR_ComTransmit(
    PduIdType ComTxPduId,
    const PduInfoType *PduInfoPtr
);
```

Figure 6.8. Typical communication stack API function example

In order for the communication stack to operate properly, adjacent modules (e.g., `Com` and the `PduR`) have to agree on the PDU handle-IDs that are exchanged between these modules. To facilitate this agreement, AUTOSAR has defined the following rules in [\[3\]](#):

- ▶ Handle-IDs can be chosen freely per module.

Thus a PDU might be sent from `Com` module to the `PduR` module with the handle-ID equal to 5. The `PduR` module itself transmits the PDU further on to the `CanIf` with the handle-ID equal to 19, and so on.

- ▶ Each PDU *must* have a unique handle-ID within the scope of the corresponding API.

For example, the `PduR` module gets transmission requests from both, the `Com` and the `Dcm` module. There are, however, two distinct APIs defined for those requests: `PduR_ComTransmit()` and `PduR_DcmTransmit()`. Therefore the `PduR` module is capable of distinguishing between two PDUs, even when they have the same handle-ID. This is so only as long as they are provided via different APIs. The interface modules (`CanIf`, `FrIf`, `LinIf`) on the other hand, provide only one API for all of their callers: `Can/Lin/FrIf_Transmit()`. In this case each PDU passed to these API functions *must* be assigned a unique handle-ID (e.g., `FrTp` and `FrNm` must not use the same handle-ID for different PDUs passed to the `FrIf`).

- ▶ Handle-IDs are defined by the module that provides the API and used by the module that calls the API.

In the case of the interaction between the `PduR` module and the `Com` module, this means that when a PDU is transferred from the `Com` module to the `PduR` module (i.e., the `Com` module calls `PduR_ComTransmit()`), the `PduR` module may define the `ComTxPduId` parameter according to its needs. Therefore the `Com`'s module configuration generator has to retrieve this parameter from the `PduR`'s configuration. When a PDU is received (i.e., the `PduR` module calls the `Com_RxIndication()`), the `Com` module may define the `ComRxPduId` parameter according to its needs. Therefore the `PduR`'s module configuration generator has to retrieve the `ComRxPduId` from the `Com`'s configuration.

- ▶ The module that provides an API may choose the handle-ID used for the invocation of the API. This means, the configuration file of the providing module contains a handle-ID value, which is used by the module that invokes the API. There might be different strategies to optimize the handle-ID values and therefore the internal structures of the implementation may have an influence on the choice of the values. In order to use the handle-IDs as index for module local table look-ups, it is often a requirement that handle-IDs must be zero-based and dense.

With the possibility to define handle-IDs, a module is now able to publish the handle-ID values to other modules. The remaining open issue is the linkage of the PDUs used by two adjacent modules. For this purpose AUTOSAR introduced the *virtual* `EcuC` module, which holds the *PDU collection* and contains several *global* PDUs. With this concept *global* PDUs are not owned by a specific module, but can be reference by any of the modules.

While a module is being configured, module-specific configuration information is created for a PDU that flows through the communication stack. Each of these *local* PDU configurations is related to the *global* PDU in the `EcuC` PDU collection, by including a reference to this *global* PDU.

In case a *module's configuration generator (MCG)* has to retrieve the handle-ID of a PDU from an adjacent module (because the adjacent module is in control of assigning the handle-ID to the PDU), the MCG follows the reference contained in the configuration of its own module to obtain the corresponding *global* PDU in the `EcuC`'s PDU collection. When this is done, the MCG browses the configuration of the adjacent modules for *local* PDUs that reference the very same *global* PDU. There should be exactly one reference to this *global* PDU from

a module which *defines* the handle-ID of this PDU. Once the *local* PDU defining the PDU's handle-ID has been determined that way, the handle-ID of this local PDU is used by the MCG when generating the configuration files for its own module.

[Figure 6.9, “PDU in the EcuC module PDU collection”](#) illustrates the *global* PDU `Pdu_Rx_CanIf_Com` in the EcuC's PDU collection.

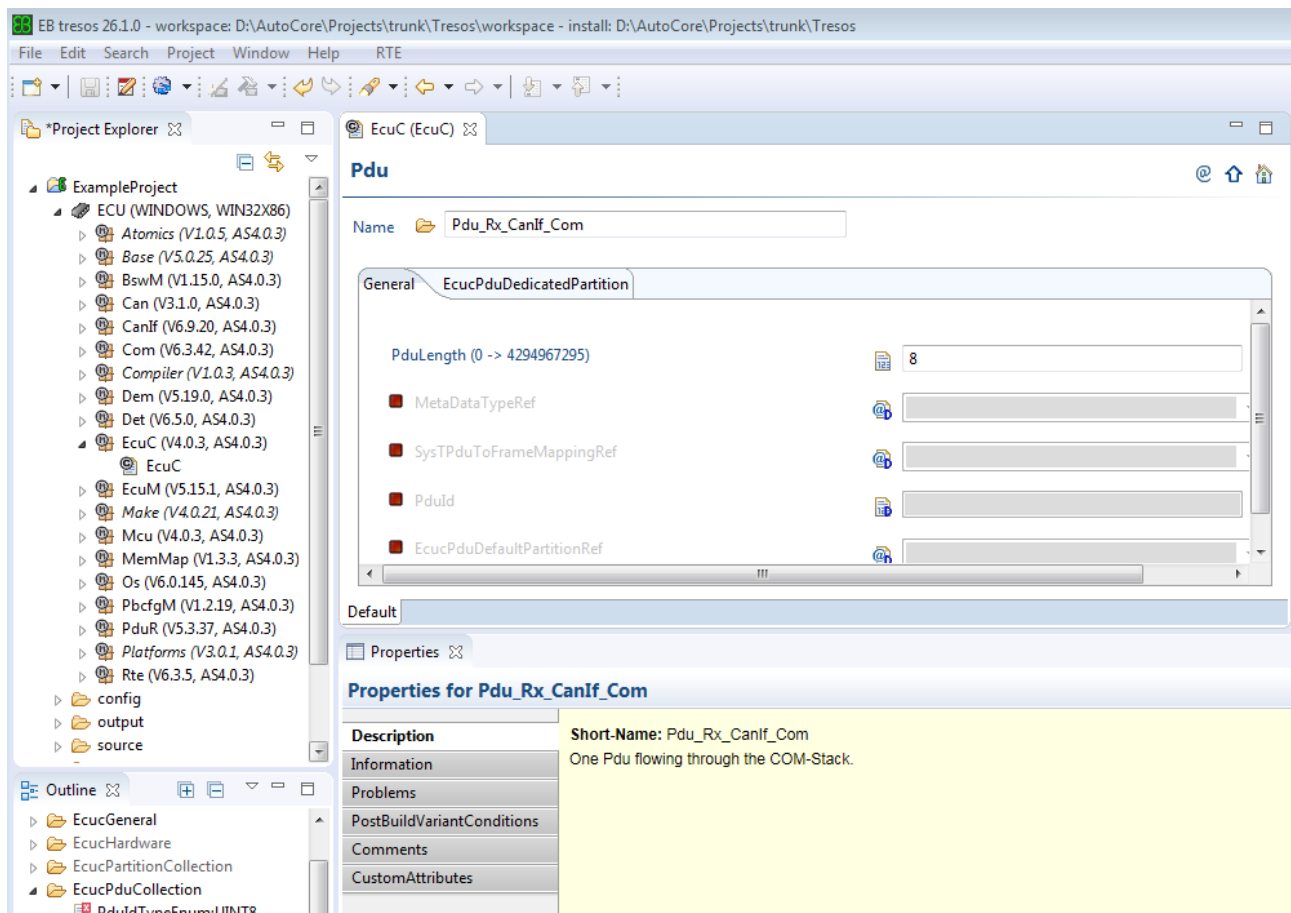


Figure 6.9. PDU in the EcuC module PDU collection

This *global* PDU is then referenced in the *local* I-PDU configuration of the `Com` module (see [Figure 6.10, “Global PDU referenced by the Com module”](#)).

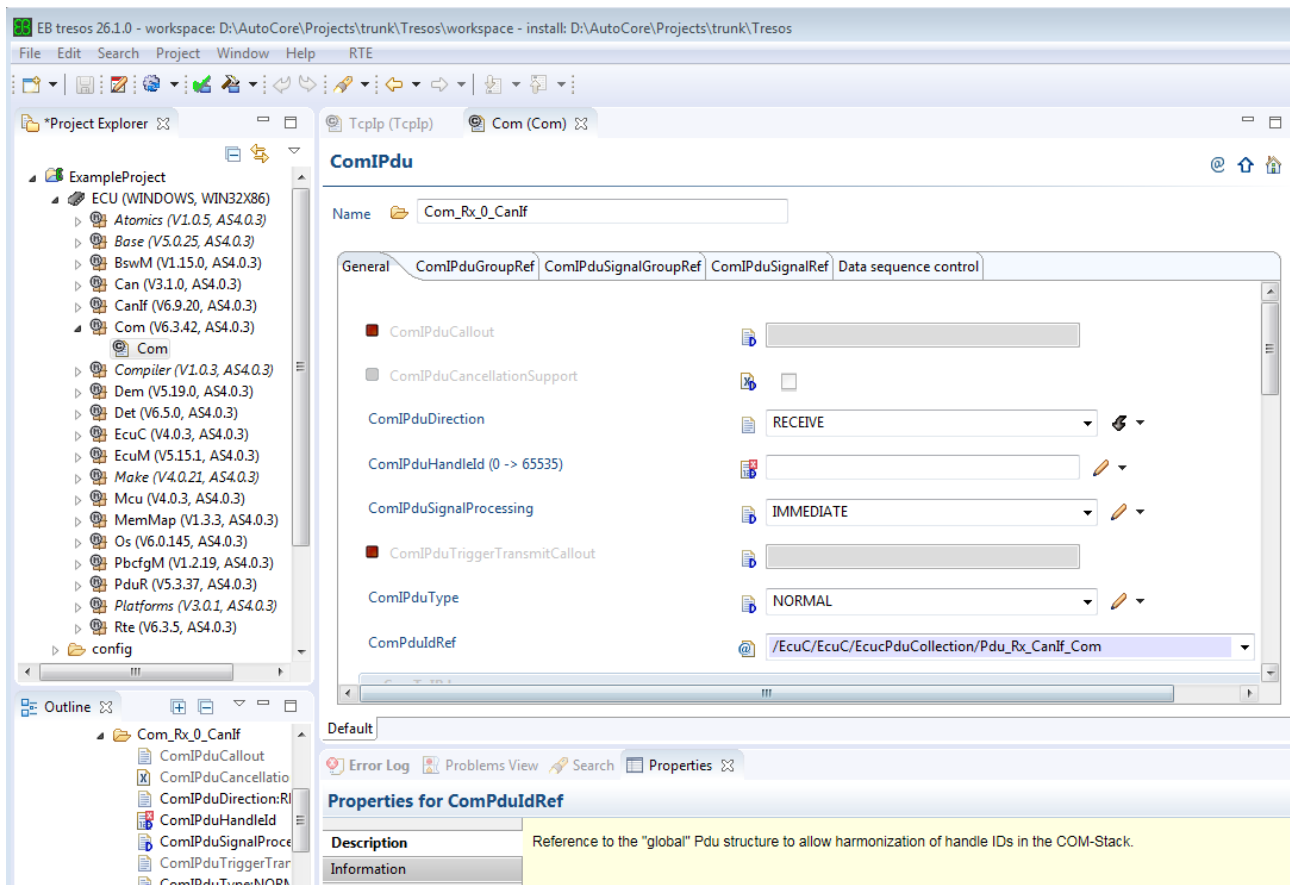


Figure 6.10. Global PDU referenced by the Com module

In a similar way the same *global* PDU is referenced in the *local* destination PDU reference of a routing path of the PduR (see [Figure 6.11, “Global PDU referenced by the PduR module”](#)).

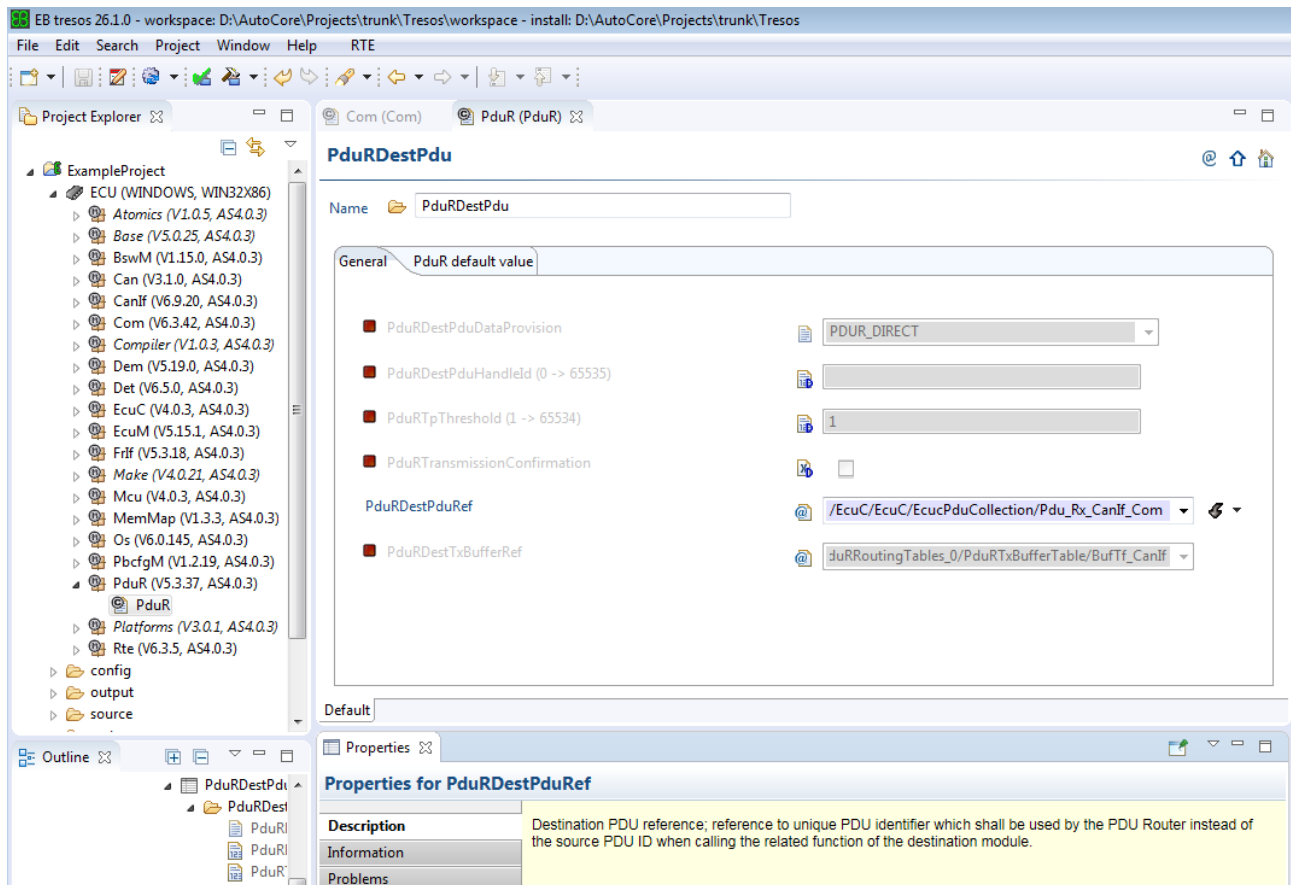


Figure 6.11. Global PDU referenced by the PduR module

Note that there is no handle-ID attribute in the destination PDU reference container of the `PduR`, since in this case, the called module is the `Com` module and thus the handle-ID attribute is defined by the `Com` module (see `ComIPduHandleId` attribute in [Figure 6.10, “Global PDU referenced by the Com module”](#)).

6.2.2.1.4. Direct vs. triggered data provision

As far as the provision of the payload data for a PDU (i. e., the SDU) is concerned, AUTOSAR distinguishes between *direct* (also called *immediate*) and *triggered* (also called *decoupled*) data provision.

[Figure 6.12, “Overview of data provision types”](#) provides an overview of these two data provision types and illustrates the calling sequences for both types using the CAN stack and the FlexRay stack as examples.

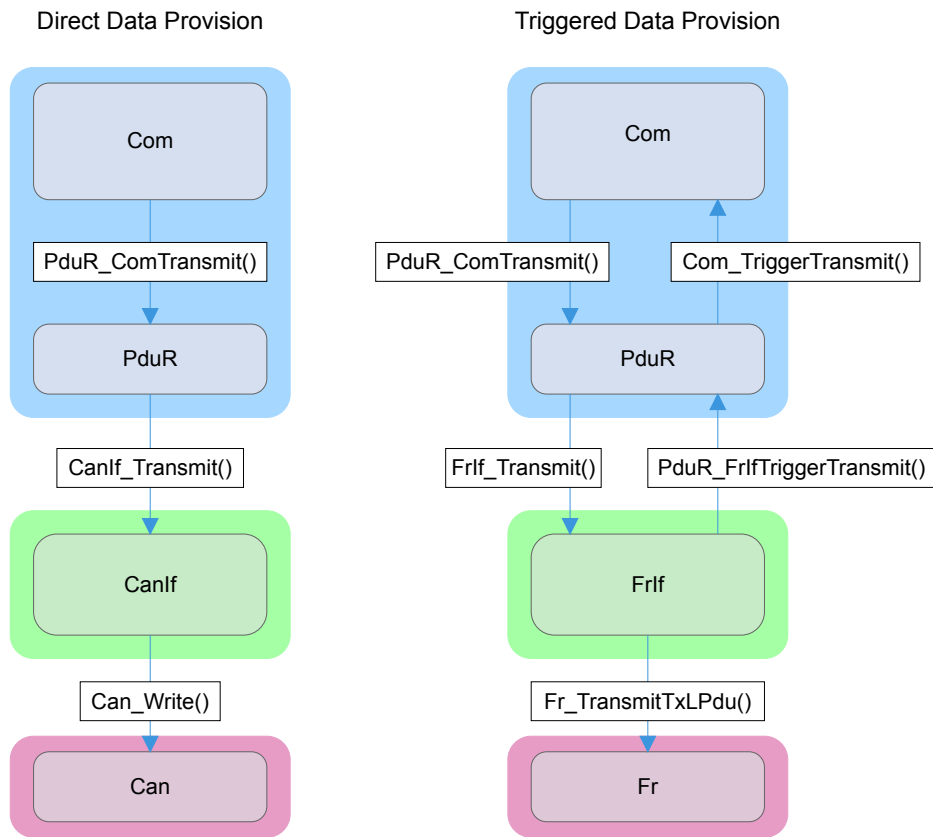


Figure 6.12. Overview of data provision types

6.2.2.1.4.1. Direct data provision

In the CAN communication stack, only direct data provision is supported. In *direct data provision*, the SDU is handed over to the underlying module via the call to the `Transmit()` function in its second parameter. This second parameter is the `PduInfoPtr` pointer, which points to a structure containing both a pointer to the SDU and the number of bytes to transmit.

[Figure 6.13, “Direct data provision in the PduR module”](#) and [Figure 6.14, “Direct data provision in the FrIf module”](#) illustrate the configuration of the `PduR` and the `FrIf` modules for direct (immediate) transmission.

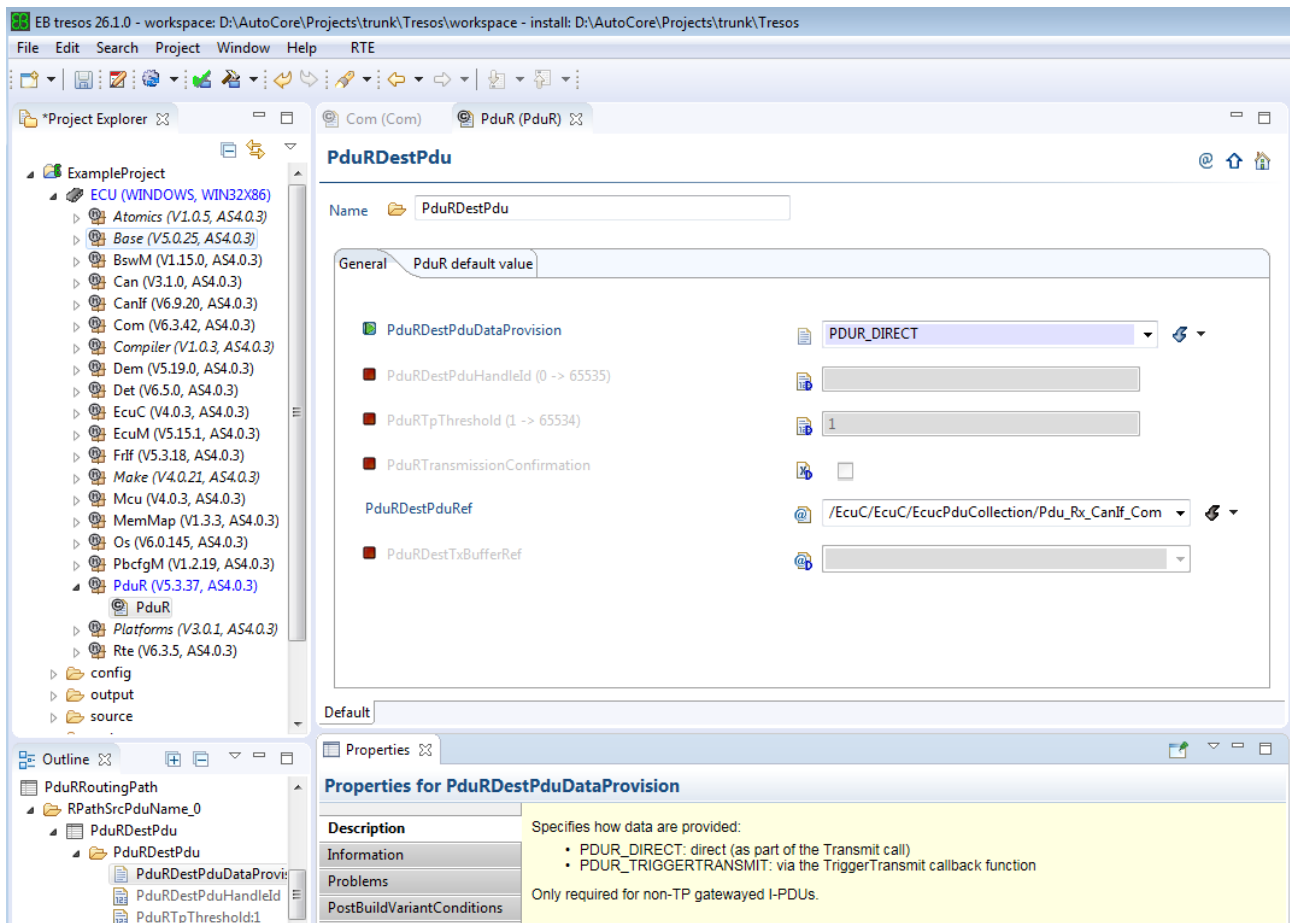


Figure 6.13. Direct data provision in the PduR module

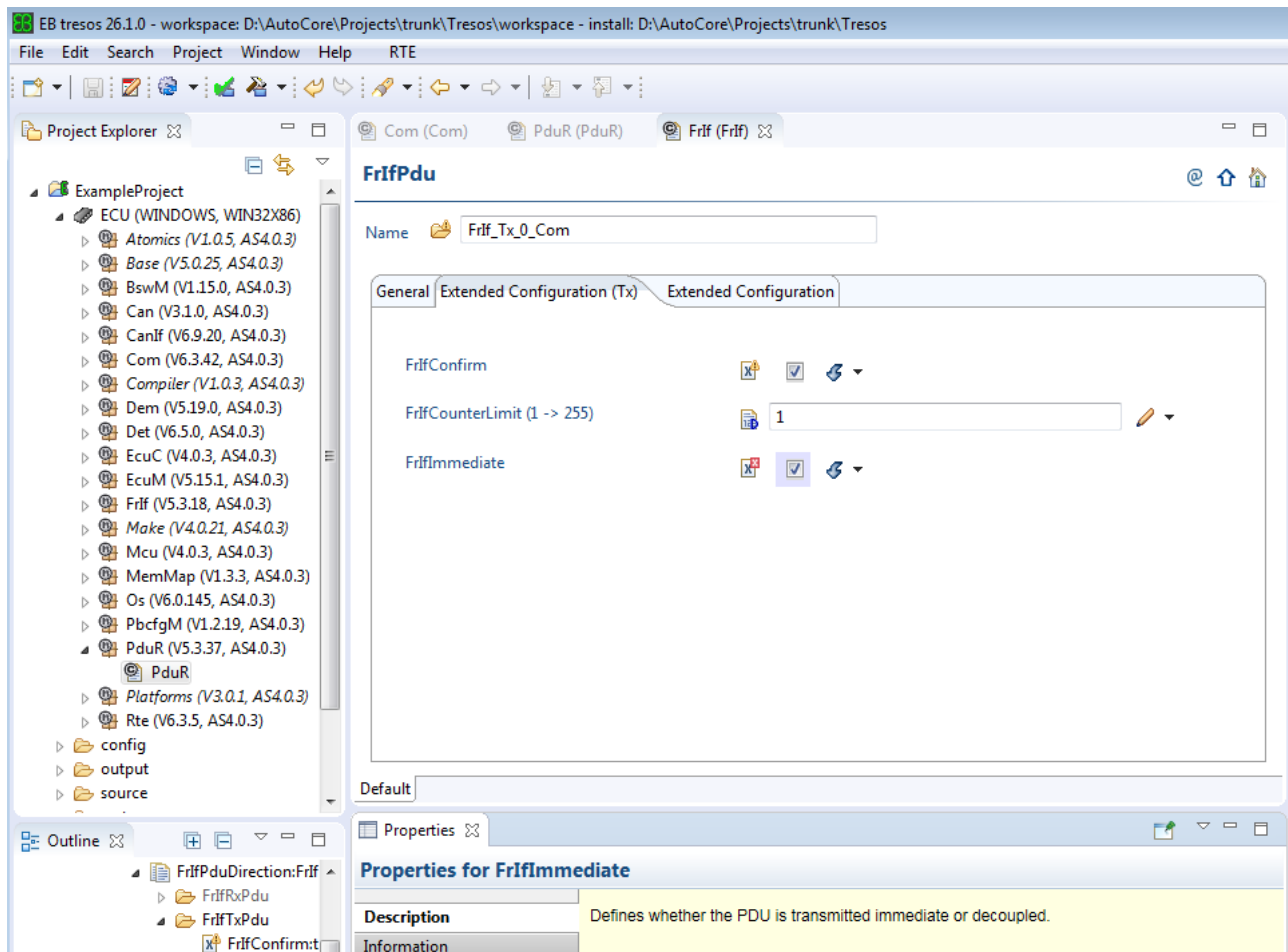


Figure 6.14. Direct data provision in the FrIf module

The type of data provision must be consistently configured in the `PduR` and the `<Net>If` modules.

6.2.2.1.4.2. Triggered data provision

The *triggered data provision* is only available with communication stacks, in which the network transmission is governed by a static cyclic communication schedule, i. e., on FlexRay and LIN. In *triggered data provision* the underlying module requests for the SDU to be provided via the `TriggerTransmit()` function. The `TriggerTransmit()` function obtains a pointer to the buffer to which the SDU will be copied to as second parameter (see [Figure 6.15, “Triggered transmission API example”](#))

```
void Std_ReturnType PduR_<User:Lo>TriggerTransmit(  
    PduIdType TxPduId,  
    PduInfoType* PduInfoPtr  
) ;
```

Figure 6.15. Triggered transmission API example

When using *triggered* data provision, the underlying module (i. e., `FrIf` or `LinIf`)

- ▶ obtains a *transmission request* via the call to the `Transmit()` function,
- ▶ internally stores the transmission request for this PDU,
- ▶ and, depending on a temporal schedule derived from the communication schedule³, calls the `TriggerTransmit()` function prior to the actual transmission of the PDU on the bus.

[Figure 6.16, “Triggered data provision in the PduR module”](#) and [Figure 6.17, “Triggered data provision in the FrIf module”](#) illustrate the configuration of the `PduR` and the `FrIf` modules for triggered (decoupled) transmission.

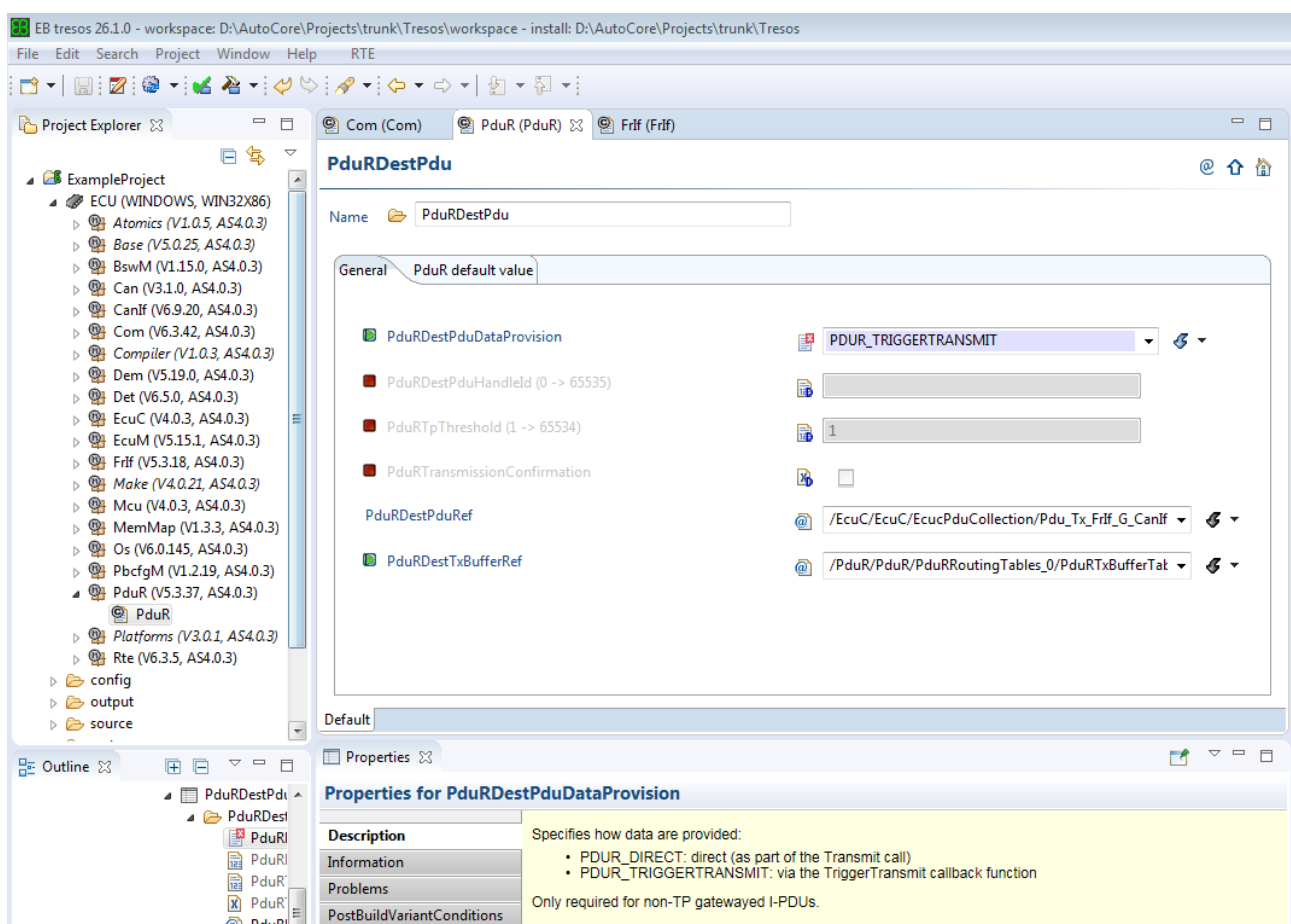


Figure 6.16. Triggered data provision in the PduR module

³In FlexRay communication this schedule is provided by the execution of the `FrIf`'s job list, which is synchronized to the FlexRay communication schedule. In LIN communication this schedule is driven by the schedule table of the `LinIf` module, which drives the communication schedule on the bus as well.

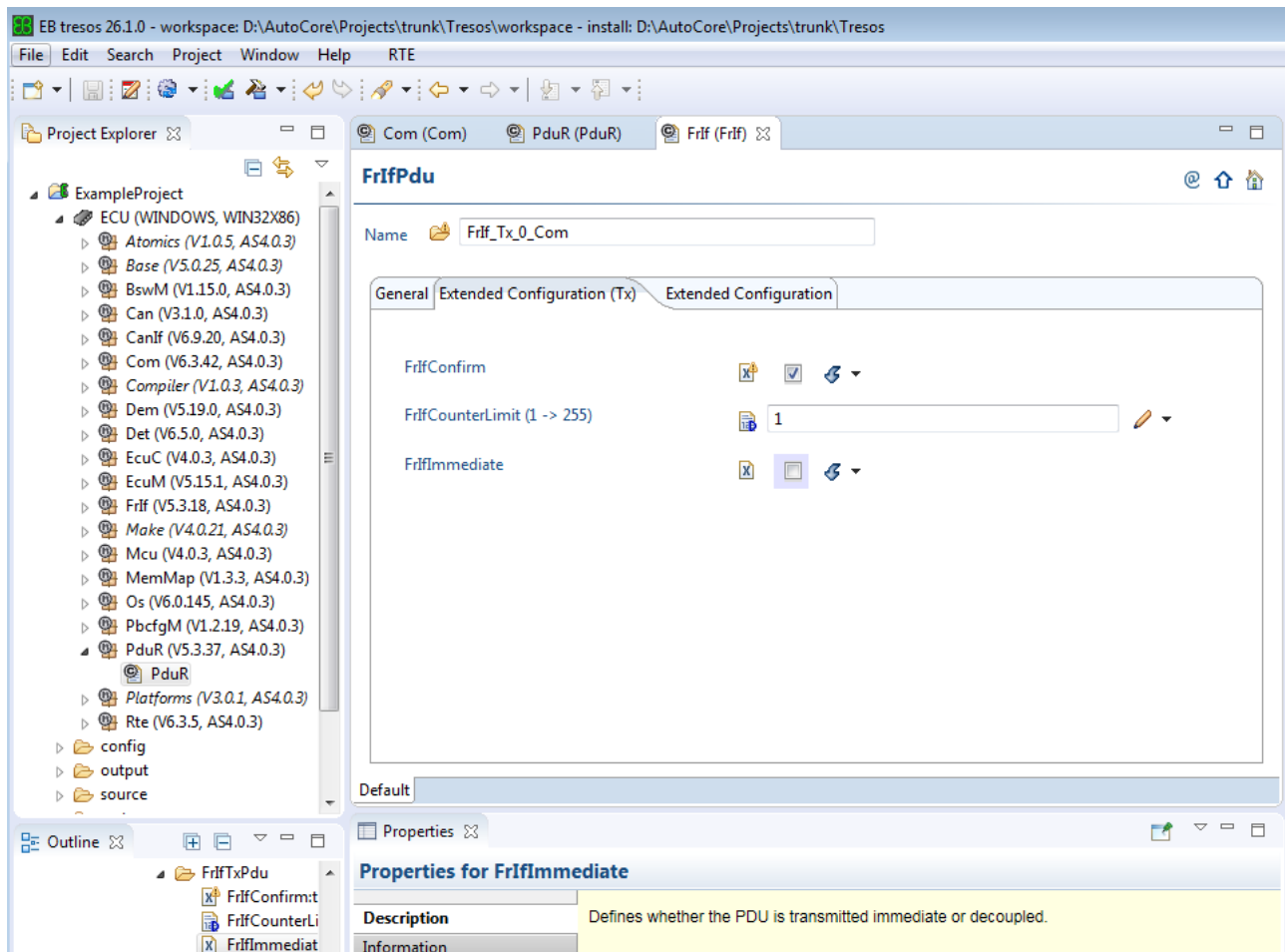


Figure 6.17. Triggered data provision in the FrIf module

6.2.2.1.5. Modules and dependencies of the network-dependent communication stack

This chapter describes the modules of the network-dependent part of the communication stack together with their dependencies. The lower modules of the communication stack depend on the used communication protocols. The lower modules are:

- ▶ the Driver module (<Net>),
- ▶ the Transceiver Driver module (<Net>Trcv),
- ▶ the Interface module (<Net>If),
- ▶ and the Transport Protocol module (<Net>Tp).

Thus AUTOSAR defines protocol-specific versions of the modules and EB tresos AutoCore implements these. The following text provides an overview of these protocol-specific modules in a generic fashion (i.e., independent of the communication protocol used). Subsequent chapters provide the protocol-specific details of these modules for each communication protocol.

Driver (<Net>):

The Driver module (`Fr`, `Can`, `Lin`) provides the basis for the respective interface module by simplifying the *transmission and the reception of frames* via the respective *communication controller (CC)*. Hereby, the Driver is designed to handle multiple CCs of the same type. Thus, if an ECU contains, for example FlexRay CCs of two different types, two different FlexRay Driver modules are required.

Transceiver Driver (<Net>Trcv):

The Transceiver Driver modules (`FrTrcv`, and `CanTrcv`) provide API functions for

- ▶ *controlling the transceiver hardware* (i.e., switching the transceivers into special modes, e.g., listen only mode)
- ▶ and for obtaining diagnostic information from the transceiver hardware (e.g., information about short circuits of the different bus lines of CAN or information about wake-up events on the bus).

In order to control the respective transceiver, the `FrTrcv` module uses the digital I/O (`Dio`) or the SPI (`Spi`) driver.

Interface (<Net>If):

Using the frame-based services provided by the Driver module, the Interface module (`FrIf`, `CanIf`, `LinIf`) takes care of the *sending and the receiving of protocol data units (PDUs)*. Hereby multiple PDUs can be packed into a single frame at the sending ECU and have to be extracted again at the receiving ECU⁴. The point in time when this packing and extracting of PDUs takes place is governed by the temporal scheduling of *communication jobs* of the FlexRay and the LIN Interface. The communication jobs of the Interface module also define the instant when the frames, which contain the packed PDUs, are handed over to the Driver module for transmission and trigger the instant when the frames are retrieved from the Driver module upon reception. In FlexRay communication, the schedule of these communication jobs is aligned with the communication schedule. In LIN communication, the schedule of the LIN Interface module governs the communication schedule on the LIN bus. In contrast to this, in CAN, the temporal schedule of the PDU transmission is governed by the `Com` module. In FlexRay, each communication job can consist of one or more communication operations. Each of these communication operations handles exactly one communication frame including the PDUs contained in this frame.

The interface modules are able to deal with multiple different drivers for different types of CCs (e.g., Freescale's MFR4300 or FlexRay CCs based on the Bosch E-Ray core). Furthermore, the Interface module wraps the API provided by the Transceiver Driver module and provides support for multiple different Transceiver Driver modules, similar to the support for multiple different Driver modules.

Driver (<Net>Tp):

The Transport protocol is used to *segment and reassemble large protocol data units (PDUs)*. In the CAN Transport protocol module (`CanTp`), this protocol is compatible to ISO TP [4] in specified settings. In the FlexRay transport protocol module (`FrTp`), the protocol is compatible to ISO TP [5]. In AUTOSAR, just as in ISO TP, the user of the services is the Diagnostic Communication Manager (`Dcm`).

⁴Currently only the FlexRay Interface module supports the packing of multiple PDUs into a single frame. For the CAN and the LIN interface modules there is a 1:1 relationship between PDUs and frames.

6.3. Network management and state management stack

6.3.1. Overview

This user guide describes the network and state management stack of EB tresos AutoCore. From this user guide you will learn about the concepts of the network and state management in AUTOSAR. You will also learn how to configure the stack.

The network and state management stack consists of

- ▶ the ComM module
- ▶ the network management modules
 - ▶ Nm
 - ▶ CanNm
 - ▶ FrNm
 - ▶ UdpNm
- ▶ the state management modules
 - ▶ CanSM
 - ▶ FrSM
 - ▶ LinSM
 - ▶ EthSM
- ▶ [Section 6.3.2, “Background information”](#) explains the concepts of the network management in AUTOSAR as well as the concept of the state management handling.
- ▶ [Section 6.3.3, “Configuring the network and state management stack”](#) provides step-by-step instructions how to configure the network and state management stack.

6.3.2. Background information

This chapter explains the concepts of the network management and state management.

The network management and state management are part of the communication stack - the control path.

The communication stack can be divided into two parts:

- **Control path:** This is the part which this user guide describes. The control path of the communication stack does not send or receive data. The only exception are the network management messages. The control path is responsible to control the state of the communication stack.
- **Data path:** The real user data are sent and received via the data path of a communication stack. SWCs use the data path of the communication stack to send data from one ECU to another. To find out details about the data path of the communication stack, refer to [Section 6.2, “Communication stacks”](#).

[Figure 6.18, “Abstracted overview of control path and data path”](#) provides a simplified overview of the modules that belong to the control path and the ones that belong to the data path.

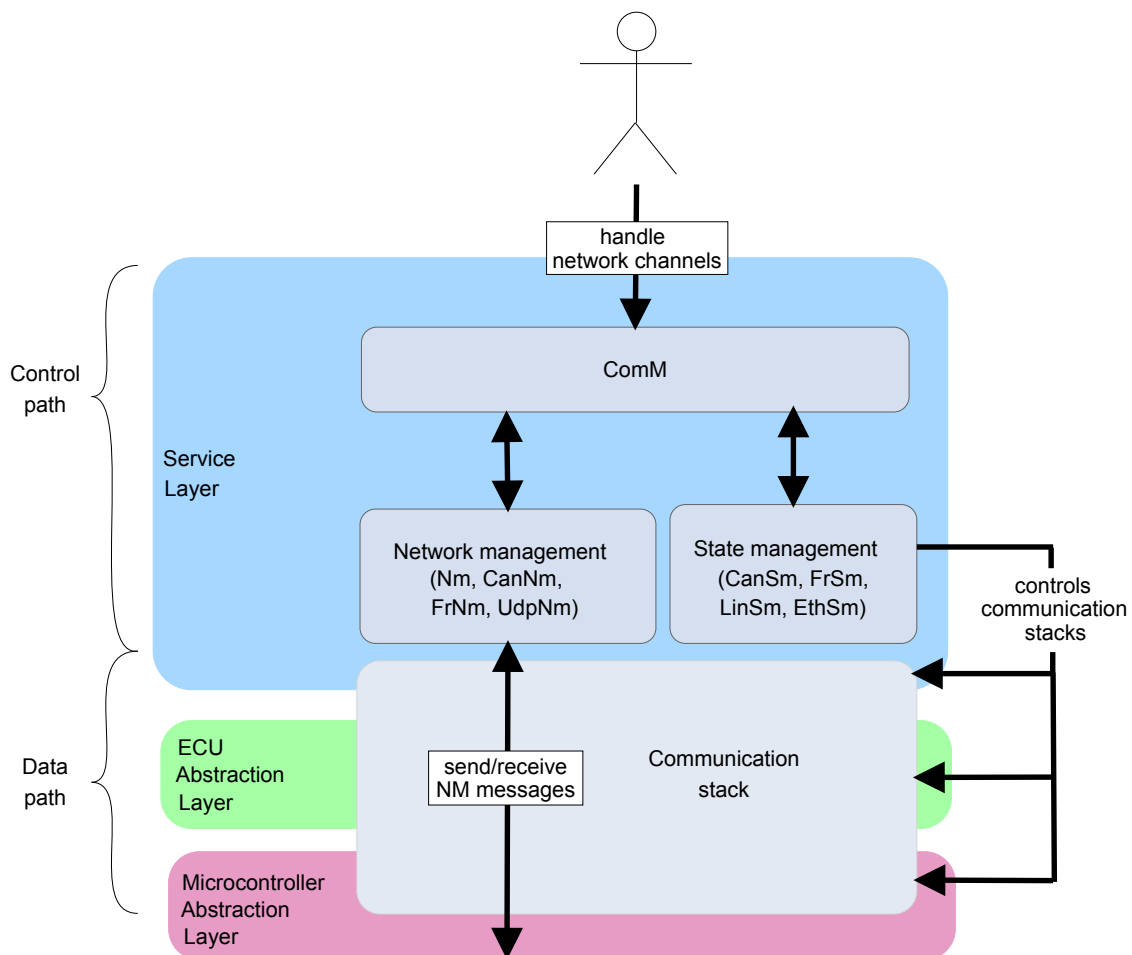


Figure 6.18. Abstracted overview of control path and data path

The modules of the control path are classified in the following logical groups:

- **Communication Manager** ComM, which simplifies the control of the complete communication stack.

For details, refer to [Section 6.3.2.1, “Network channels and states”](#).

- **Bus-dependent network management modules:** `CanNm`, `FrNm`, `UdpNm`. These modules implement the bus-specific network management.

For details, refer to [Section 6.3.2.2, “Network management”](#).

- **Bus-independent network management module:** `Nm`. The `Nm` module implements:
 - A bus-independent adaptation layer.
 - A coordinator algorithm, which allows to put several connected networks into sleep state synchronously.

For details, refer to [Section 6.3.2.2, “Network management”](#).

- **State manager modules:** `CanSM`, `FrSM`, `LinSM`, `EthSM`. These modules implement the bus-specific start-up and shut-down.

For details, refer to [Section 6.3.2.3, “State management”](#).

[Figure 6.19, “Overview of the communication stack control paths”](#) provides a detailed overview of all modules, which belong to the control path of an AUTOSAR communication stack.

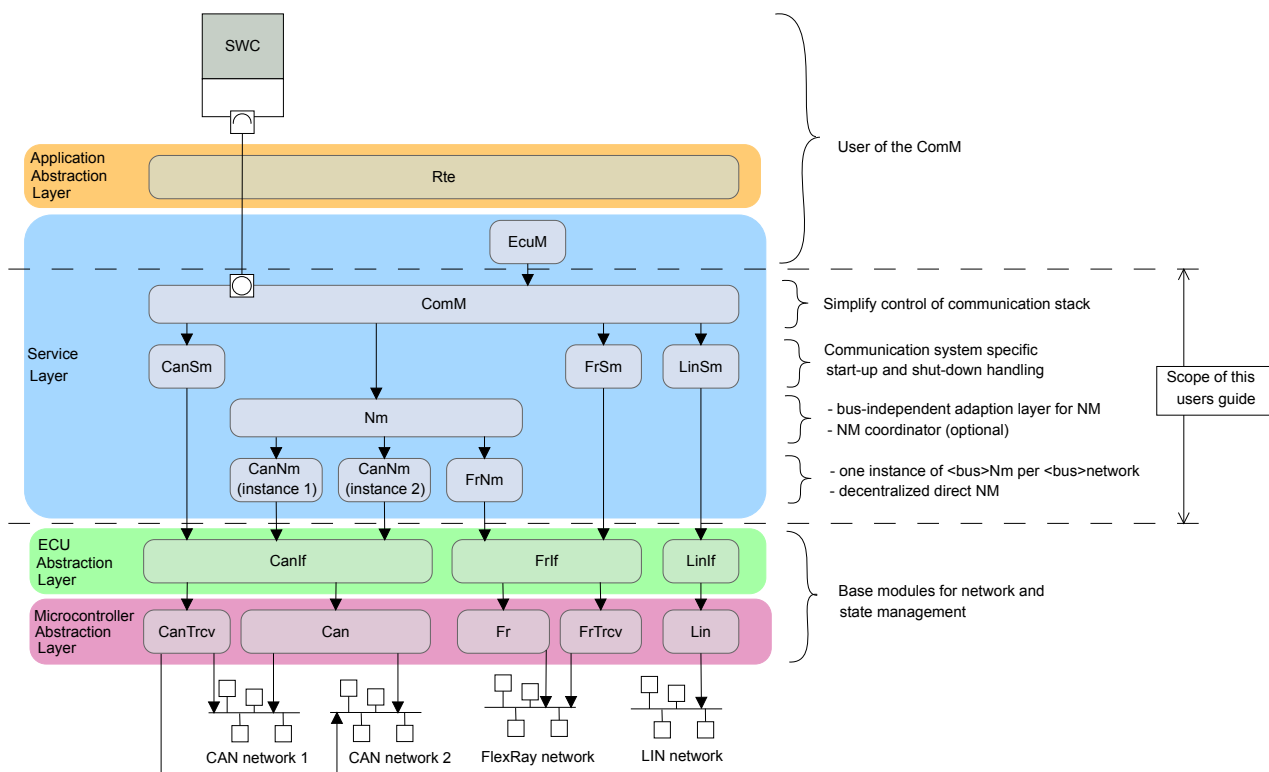


Figure 6.19. Overview of the communication stack control paths

6.3.2.1. Network channels and states

The `ComM` module provides an abstraction of the controlled communication system. The user configures and accesses *network channels*. The network channels are independent of the underlying communication system (CAN, FlexRay, UDP/IP/Ethernet or LIN).

The network channels are accessed by two users:

1. `EcuM/BswM`, which request communication e.g. after a wakeup
2. Software components

For each network channel, there is an specific state machine within the `ComM` module. The state machine has the following states:

- ▶ Full communication
 - ▶ Network requested
 - ▶ Ready sleep
- ▶ Silent communication
- ▶ No communication

The communication behavior differs for each state. This is shown in the table below:

State	Message transmis- sion	Message reception	Network manage- ment/bus commu- nication	Wake-up capability
Full communica- tion/network re- quested	On	On	Requested	Not applicable
Full communica- tion/ready sleep	On	On	Released	Not applicable
Silent communica- tion	Off	On	Released	<ul style="list-style-type: none"> ▶ User request ▶ Network indica- tion
No communication	Off	Off	Released	<ul style="list-style-type: none"> ▶ User request ▶ Passive wake- up

Table 6.1. States and communication behavior

The following example shows the request of full communication for a CAN stack:

User --> `ComM`: requests communication:

1. ComM --> CanSM: requests full communication from CanSM
 - a. CanSM --> CanIf: sets transceiver mode to normal
 - i. CanIf --> CanTrcv: sets transceiver mode to normal
 - b. CanSM --> CanIf: sets controller mode to started
 - i. CanIf --> Can: sets controller mode to started
 - A. Can --> hardware: starts CAN controller
 - c. CanSM --> Com: starts Rx I-PDU groups for CAN
 - d. CanSM --> Com: starts Tx I-PDU groups for CAN
 - e. CanSM --> ComM: indicates switch to full communication
2. ComM --> Nm: requests network from Nm
 - a. Nm --> CanNm: requests network from CanNm
 - i. CanNm: switches to network mode and starts transmission of NM messages

6.3.2.2. Network management

The network management stack consists of:

- ▶ the bus-independent network management interface module Nm,
- ▶ the bus-dependent CAN network management module CanNm
- ▶ the bus-dependent FlexRay network management module FrNm
- ▶ the bus-dependent UDP/IP/Ethernet network management module UdpNm

There is no network management for LIN.

The purpose of the Nm module is to provide a bus-independent interface towards the ComM module. Furthermore the Nm can be configured as *NM coordinator*. The NM coordinator handles the synchronous shut down if two or more buses are connected.

6.3.2.2.1. Network management algorithm

The AUTOSAR network management algorithm for CAN, UDP/IP/Ethernet as well as for FlexRay is based on a decentralized direct network management strategy. This means, the same algorithm runs on each node. The synchronization is done via network management messages. The transmission of network management messages is issued by the CanNm, UdpNm respectively the FrNm module. The bus-dependent network man-

agement modules use the communication stack via the interface modules (`CanIf`, `SoAd`, `FrIf`) to transmit and receive the network management messages. For an illustration of this, refer to [Figure 6.19, “Overview of the communication stack control paths”](#).

NOTE



No token ring for network management is built up

The AUTOSAR network management does not build up a token ring like OSEK network management does. The completely different algorithm *decentralized direct network management* is used instead.

If a network node requires bus communication, it transmits network management messages.

If a network node does not require bus communication any more, it stops transmitting network management messages. The bus communication is released.

If the bus communication is released and there are no network management messages on the bus, the nodes switch to the bus sleep mode state. The different states are explained in more detail in [Section 6.3.2.2.2, “Network management states”](#).

6.3.2.2.2. Network management states

The network management algorithm within the `CanNm`, `UdpNm` respectively the `FrNm` module handles the states according to [Figure 6.20, “The network management states”](#).

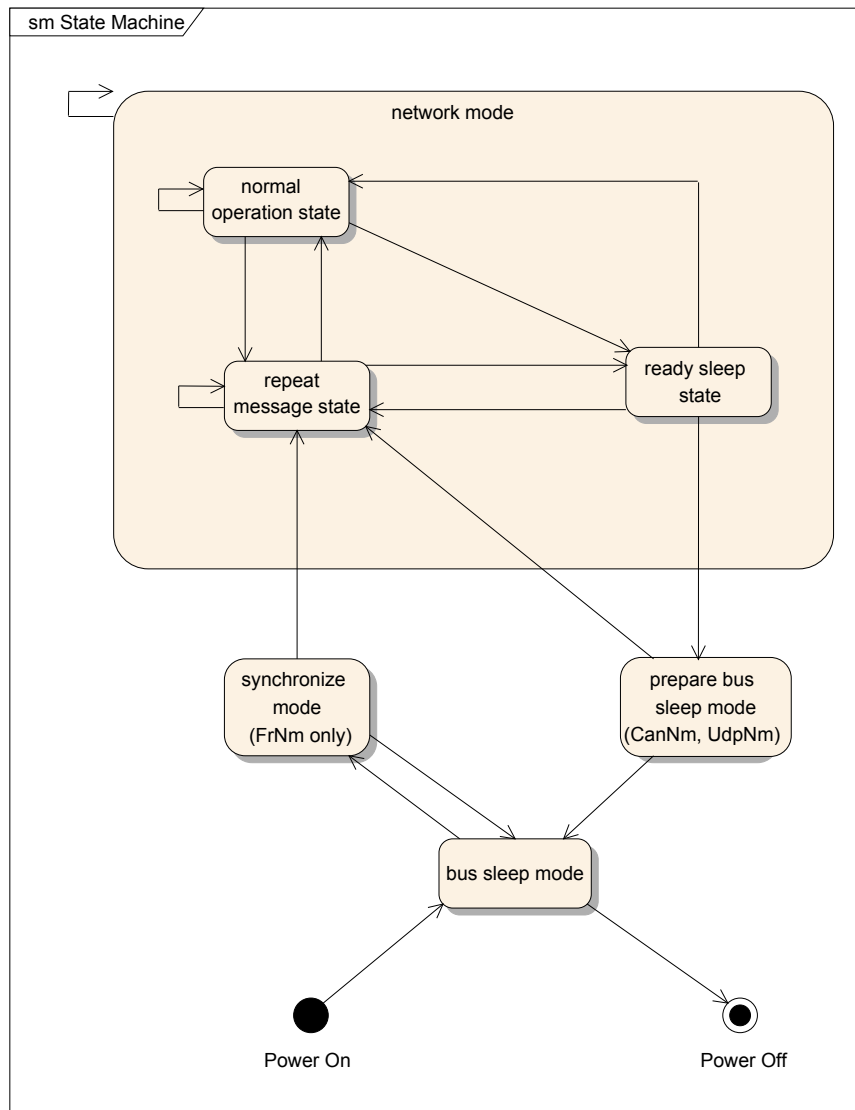


Figure 6.20. The network management states

The states/modes⁵ have the following purposes:

► *bus sleep mode*

The Bus sleep mode is the default mode on the start of the network management state machine, where it remains unless the NM is started. In Bus Sleep Mode the communication controller can be switched into the sleep mode where wakeup mechanisms are activated and power consumption is reduced to a minimal level.

► *synchronize mode* (FrNm only)

⁵AUTOSAR uses the notation mode for a state and the notation state for a sub-state

In the synchronize mode the `FrNm` state machine waits to be synchronized to the `FrNm` repetition cycle. This is necessary as the FlexRay NM is dependent on state changes being synchronized across the NM-cluster.

► *prepare bus sleep mode* (`CanNm` and `UdpNm`)

In the prepare bus sleep mode, the bus activity is calmed down (i.e. queued messages are transmitted in order to make all Tx-buffers empty) and finally there is no activity on the bus in the prepare bus sleep mode.

► *network mode*

If there is at least one ECU within the network which requests communication, the network mode is entered. The network mode comprises of the following states:

► *repeat message state*

The repeat message state is entered always when entering the network mode. As a NM message is sent out within this state, the repeat message state ensures, that any transition from bus sleep to the network mode becomes visible to the other nodes on the network.

► *normal operation state*

The normal operation state ensures that any node can keep the NM-cluster awake as long as the network is requested.

► *ready sleep state*

The ready sleep state ensures that any node in the NM-cluster waits to transition to the Bus-Sleep Mode as long as any other node keeps the NM-cluster awake.

Within the ready sleep state, the node does not send out NM messages any more, but it receives NM messages from other nodes.

The dependency between the network management algorithm (see [Section 6.3.2.2.1, “Network management algorithm”](#)) and the network management states is shown in the table below:

Network management state	Local ECU state	Local ECU transmits NM messages	NM messages on the bus
bus sleep	power on	no	don't care
network mode: repeat message	request communication	yes	don't care
network mode: normal operation	request communication	yes	don't care
network mode: ready sleep	release communication	no	yes
prepare bus sleep	release communication	no	no

Network management state	Local ECU state	Local ECU transmits NM messages	NM messages on the bus
bus sleep	sleep	no	no

Table 6.2. Network management states and communication behavior

6.3.2.2.3. Structure of an NM PDU

In [Figure 6.21, “Structure of an NM PDU”](#) is presented the default structure of the NM PDU. The location of the source node identifier or of the control bit vector could be different depending on the bus that is used or depending on the configuration.

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte n	User data n							
...	...							
Byte 5	User data 3							
Byte 4	User data 2							
Byte 3	User data 1							
Byte 2	User Data 0							
Byte 1	Control Bit Vector (default)							
Byte 0	Source Node Identifier (default)							

Figure 6.21. Structure of an NM PDU

- ▶ Source Node Identifier: stores the Node Identifier configured in parameter UDPNM_NODE_ID
- ▶ Control Bit Vector(CBV): set of control flags
- ▶ User Data 0 - n: set of bytes used for certain features or for OEM specific extensions

NOTE

For `CanNm`, `FrNm` the maximum number of User Data bytes is 6.



For `UdpNm` the maximum number of User Data bytes shall not exceed Maximum Transmission Unit(MTU).

6.3.2.2.3.1. Control bit vector

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Byte 1	Not available	PNI Bit	Reserved	Active Wakeup Bit	NM Coordinator Sleep Ready	Reserved	Reserved	Repeat Message Request

Figure 6.22. Structure of control bit vector

- ▶ Bit 0: Repeat Message Request
 - ▶ 0: Repeat Message State not requested
 - ▶ 1: Repeat Message State requested
- ▶ Bit 3 :NM Coordinator Sleep Bit
 - ▶ 0: Start of synchronized shutdown is not requested by main coordinator
 - ▶ 1: Start of synchronized shutdown is requested by main coordinator
- ▶ Bit 4 Active Wakeup Bit
 - ▶ 0: Node has not woken up the network (passive wakeup)
 - ▶ 1: Node has woken up the network (active Wakeup)
- ▶ Bit 6 Partial Network Information Bit (PNI)
 - ▶ 0: NM message contains no Partial Network request information
 - ▶ 1: NM message contains Partial Network request information
- ▶ Bit 1, 2, 5, 7 are reserved for future extensions

6.3.2.2.3.2. NM user data

The user data that is part of the NM PDU represents supplementary application specific data that is attached to every NM message sent on the bus. User data in NM PDU is also used for realising partial networking functionality or for sending network management state information.

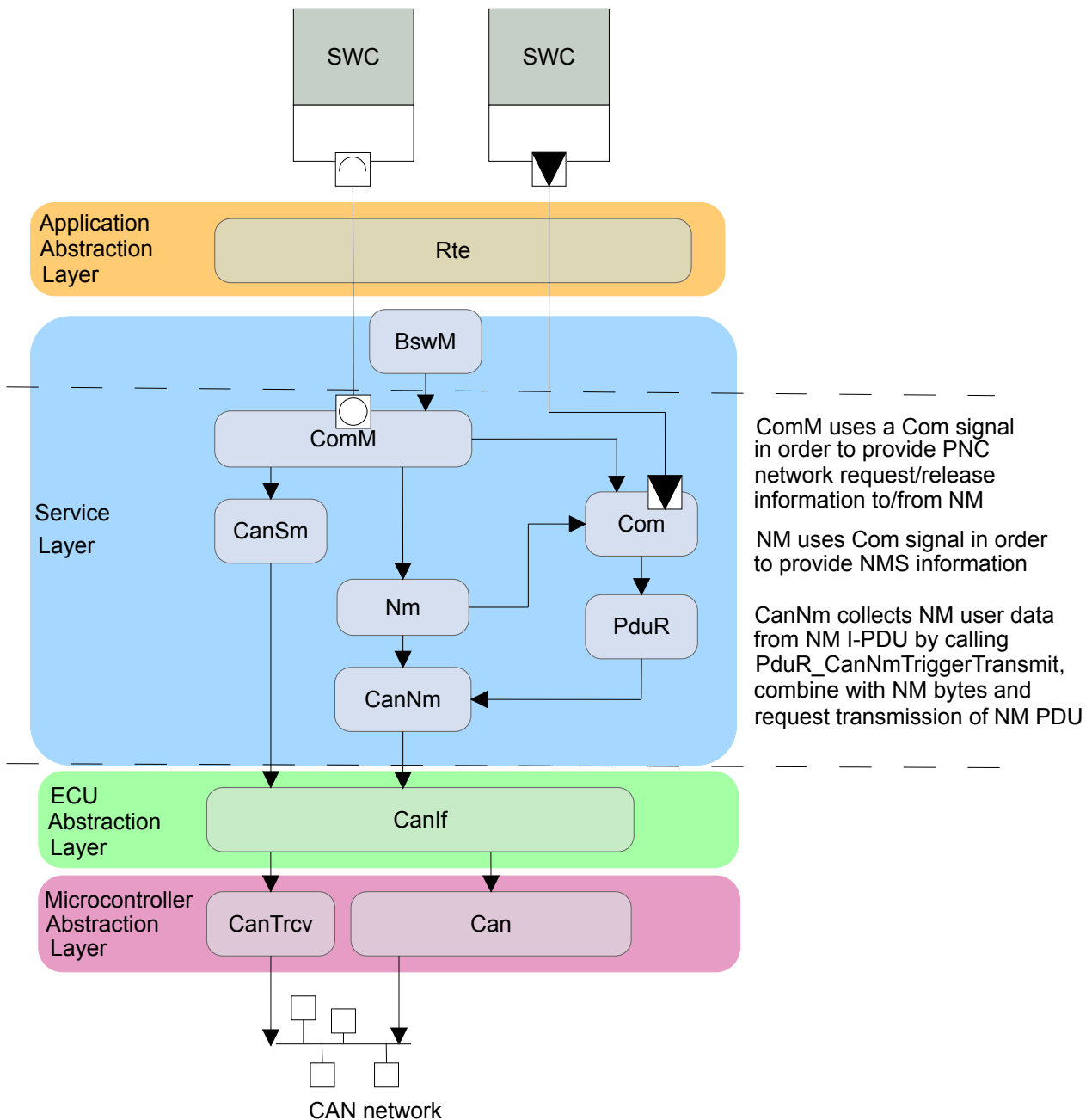


Figure 6.23. NM user data

AUTOSAR offers two alternatives for transmitting NM user data:

- NM user data is written via the `Nm_SetUserData` interface. For this purpose the bus specific `<Bus>Nm_SetUserData` shall be called by NM.

- ▶ NM user data is accessed using the communication stack. For this purpose you must configure `Com` signals and aggregate them in I-PDUs. The following users could call the function `Com_SendSignal()` in order to provide NM user data:

- ▶ NM module to send NMS information to the `Com`,
- ▶ `ComM` module to send partial networking information to the `Com`,
- ▶ The application to send additional information e.g. wakeup reasons to the user data I-PDU,

The different user data signals are collected within the `Com` module and are sent via a trigger transmit mechanism from the `Com` to the `<Bus>Nm` module.

6.3.2.3. State management

As shown in [Figure 6.19, "Overview of the communication stack control paths"](#), there are the following state management modules:

- ▶ `CanSM`
- ▶ `FrSM`
- ▶ `LinSM`
- ▶ `EthSM`

The state manager modules perform the following tasks:

- ▶ provides bus-independent interface towards the `ComM` module
- ▶ handles bus-specific wakeup
- ▶ sets transceiver mode (CAN and FlexRay only)
- ▶ sets controller mode (CAN and FlexRay only)
- ▶ starts associated PDU groups in the `CanIf` module (CAN only)
- ▶ bus-specific go to sleep
- ▶ switches schedule table (LIN only)
- ▶ Reports state transitions to `BswM` module.

6.3.2.4. Module dependencies

- ▶ **State manager modules** (`CanSM`, `FrSM`, `LinSM`, `EthSM`): The state manager modules have the following dependencies:
 - ▶ Associated interface module. For example the `CanIf` module is needed by the `CanSM` module for setting controller and transceiver modes.

- ▶ Communication manager module `ComM`, e.g. for indicating requested bus modes.
- ▶ Standard dependencies. This means the `SchM` is needed to realize critical sections, the `Det` is needed for reporting development errors and the `Dem` module is needed for reporting diagnostic events.
- ▶ **Bus-dependent network management modules** `CanNm`, `FrNm`, `UdpNm`. The bus-dependent network management modules have the following dependencies:.
 - ▶ Associated interface module(`FrIf` for `FrNm`, `CanIf` for `CanNm` and `SoAd` for `UdpNm`). For example the `CanIf` module is needed by the `CanNm` module for sending and receiving network management messages.
 - ▶ Network management module `Nm`, e.g. for indicating changes in the network state.
 - ▶ Standard dependencies, that means the `SchM` is needed to realize critical sections, the `Det` is needed for reporting development errors and the `Dem` module is needed for reporting diagnostic events.
- ▶ **Bus independent network management module** `Nm` has the following dependencies:
 - ▶ Bus-dependent network management modules.
 - ▶ Communication manager module.
 - ▶ Standard dependencies. This means the `SchM` is needed to realize critical sections, the `Det` is needed for reporting development errors and the `Dem` module is needed for reporting diagnostic events.
- ▶ **Communication Manager** `ComM` has the following dependencies:
 - ▶ Bus-independent network management module `Nm`, e.g. for receiving wakeup indications.
 - ▶ Diagnostic communication manager module `Dcm`.
 - ▶ ECU state manager module `EcuM`, e.g. for receiving the requested mode or for receiving wake-up indications.
 - ▶ NVRAM manager module `NvM`
 - ▶ Run-time environment `Rte` to propagate the user requests from SWCs and to indicate mode changes to users.
 - ▶ Bus-dependent state management modules `CanSM`, `FrSM` and `LinSM` to receive the bus modes.
 - ▶ Standard dependencies, that means the `SchM` is needed to realize critical sections, the `Det` is needed for reporting development errors and the `Dem` module is needed for reporting diagnostic events.

6.3.3. Configuring the network and state management stack

This chapter provides step-by-step instructions for configuring the network and state management stack.

Please note that the OEM defines the timing parameters and the `<Bus>Nm` has to be configured according to OEM criteria.

To achieve the best results, proceed through this chapter in the order suggested. If you follow the configuration order below, you can set the references inside all module configurations without needing to switch between module configurations:

1. First of all, configure the modules listed in [Section 6.3.3.1, “Conditions for configuring the network and state management stack”](#).
2. Then configure the network and state management modules in this order:
 - ▶ [Section 6.3.3.2, “Configuring the ComM module”](#)
 - ▶ [Section 6.3.3.3, “Configuring the Nm module”](#)
 - ▶ [Section 6.3.3.4, “Configuring the bus-dependent network management modules”](#)

The configuration example is based on [Figure 6.19, “Overview of the communication stack control paths”](#). This means the following buses are connected:

- ▶ two CAN networks
- ▶ one FlexRay network
- ▶ one LIN network


6.3.3.1. Conditions for configuring the network and state management stack

The network and state management configuration references the following modules. To be able to set these references, configure the following modules first:

- ▶ **EcuC** module. The **EcuC** module contains a list of global PDUs. The **CanNm**, **FrNm** and **UdpNm** module send and receive network management PDUs and have therefore references into the **EcuC** module
- ▶ **CanIf** module. The **CanIf** controllers and transceivers are referenced by the **CanSM** module
- ▶ **FrIf** module. The **FrIf** clusters are referenced by the **FrSM** module
- ▶ **LinIf** module. The **LinIf** schedule tables are referenced by the **LinSM** module
- ▶ **EthIf** module. The **EthIf** controllers are referenced by the **EthSM** module

6.3.3.2. Configuring the ComM module

To configure the **ComM**:

- ▶ Open the **ComM** module configuration.
- ▶ Switch to the tab **Users**.
- ▶ Configure a user for each software component, which uses the **ComM** services: Click the  button. The **User ID** is calculated automatically.

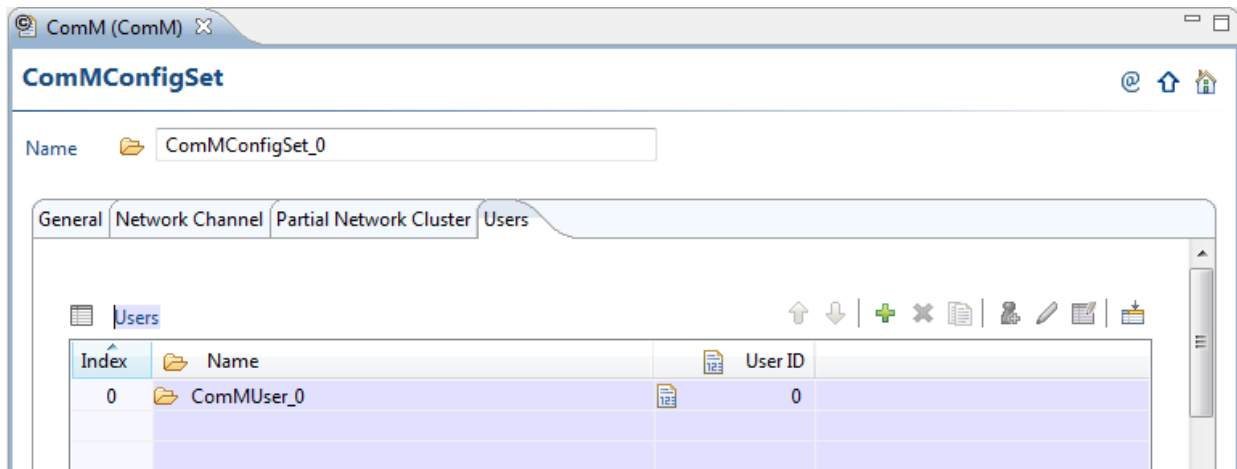


Figure 6.24. ComM tab User

- ▶ Switch to the tab **Partial Network Cluster**.
- ▶ Configure a PNC for each group of channels.

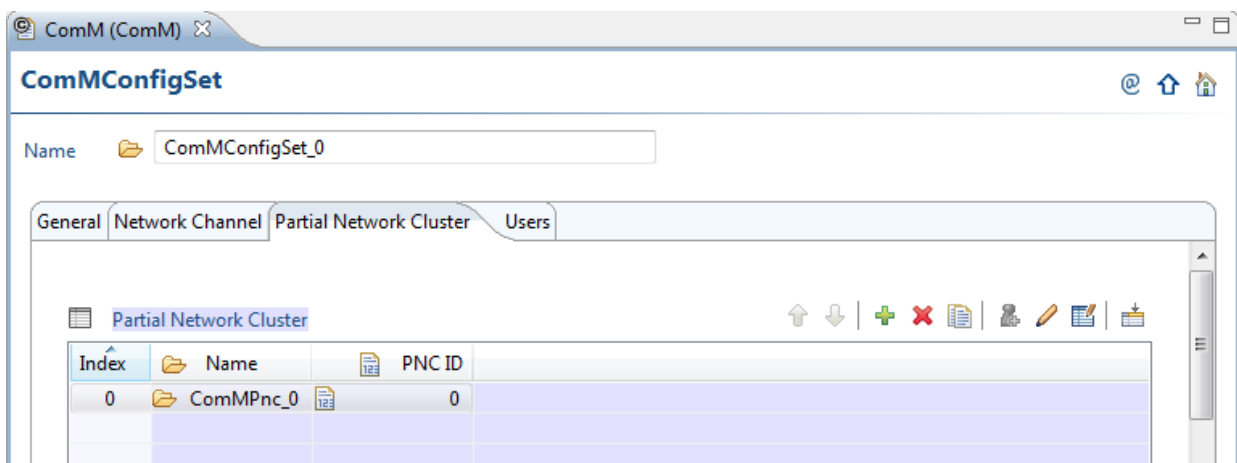


Figure 6.25. ComM tab Partial Network Cluster

- ▶ Switch to the tab **Network Channel**.
- ▶ Configure a network channel for each bus connected to the ECU.

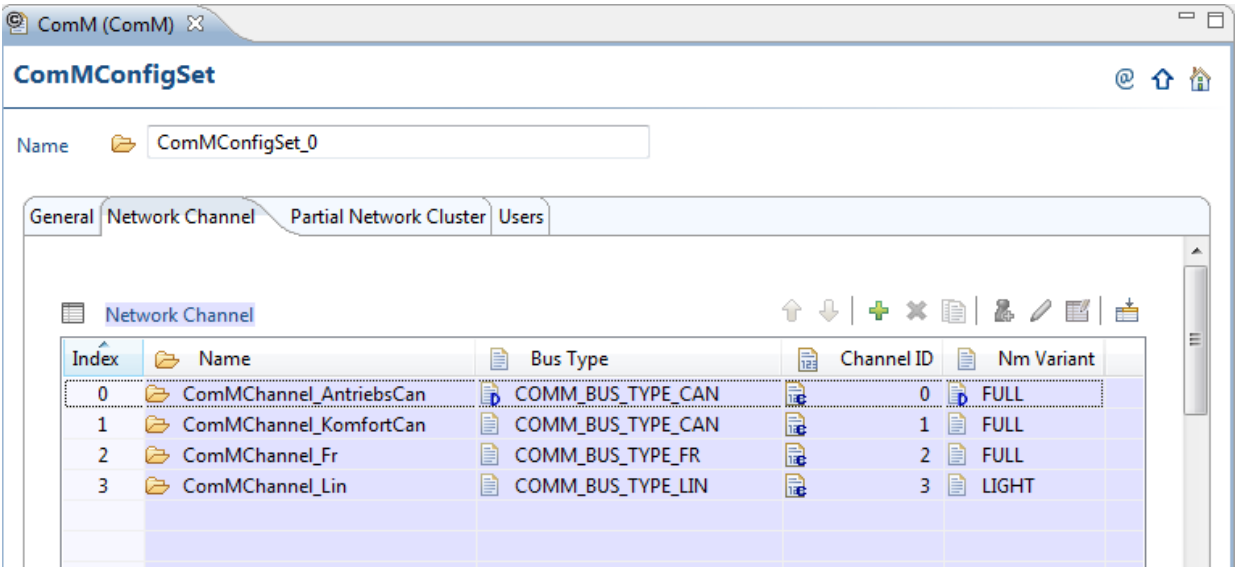


Figure 6.26. ComM tab Network Channel

The configuration displayed in the image works for an ECU with two connected CAN buses, one LIN bus and one FlexRay bus.

For a detailed description of all configuration parameters, refer to the document EB tresos AutoCore module references.

6.3.3.3. Configuring the Nm module

To configure the Nm module:

- ▶ Open the Nm module configuration.
- ▶ Switch to the tab **List of Nm Channels**.

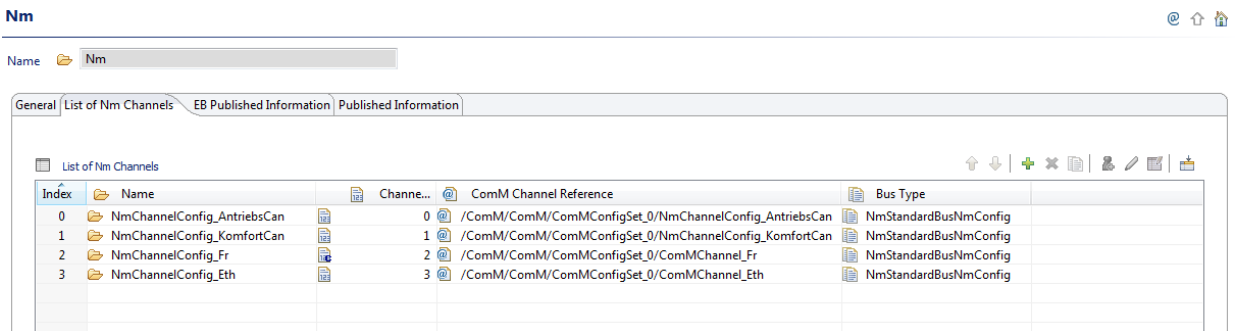


Figure 6.27. Nm interface tab List of Nm channels

- ▶ Add a network channel for each connected CAN, UDP/IP/Ethernet or FlexRay bus: Click the **+** button.

For a detailed description of all configuration parameters, refer to the document EB tresos AutoCore module references.

6.3.3.4. Configuring the bus-dependent network management modules

To configure the bus-dependent network management modules:

- ▶ Open the `CanNm` module configuration.
- ▶ Switch to the tab **CanNmGlobalConfig**.
- ▶ Select the multiple configuration container `CanNmGlobalConfig_0`.
- ▶ Switch to the tab **Channel Configuration**.

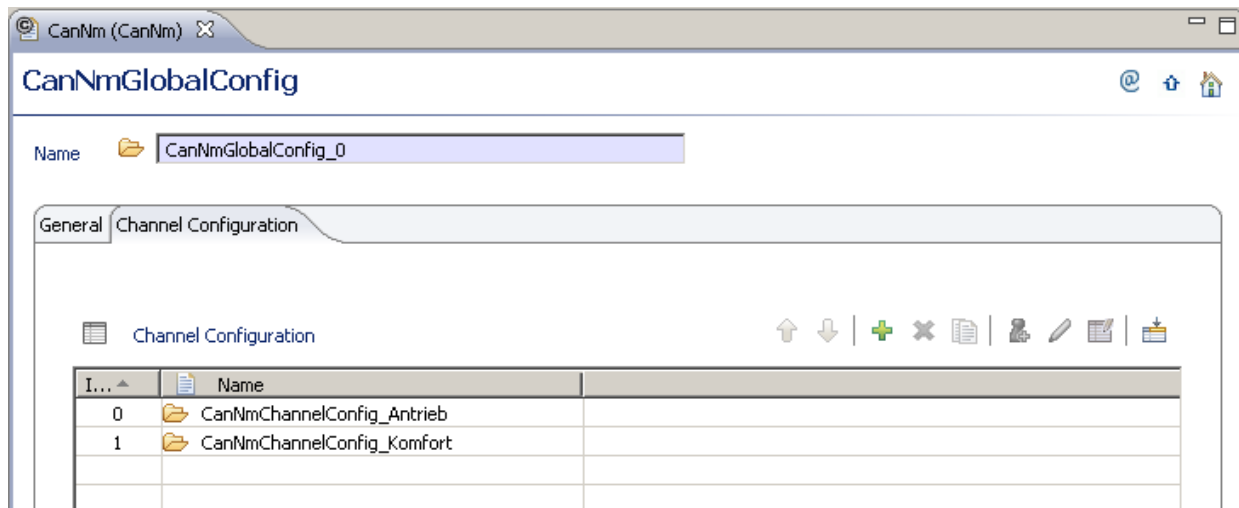


Figure 6.28. CanNm tab channel configuration

- ▶ Configure one channel for each connected CAN bus by clicking the **+** button.

For a detailed description of all configuration parameters, refer to the document EB tresos AutoCore module references.

6.3.3.5. Configuring the bus-dependent state management modules

To configure the bus-dependent state management modules:

- ▶ Open the `CanSM` module configuration.
- ▶ Switch to the tab **List of Configurations**.
- ▶ Select the multiple configuration container `CanSM_Config_0`.

- Switch to the tab **List of Can Networks**.

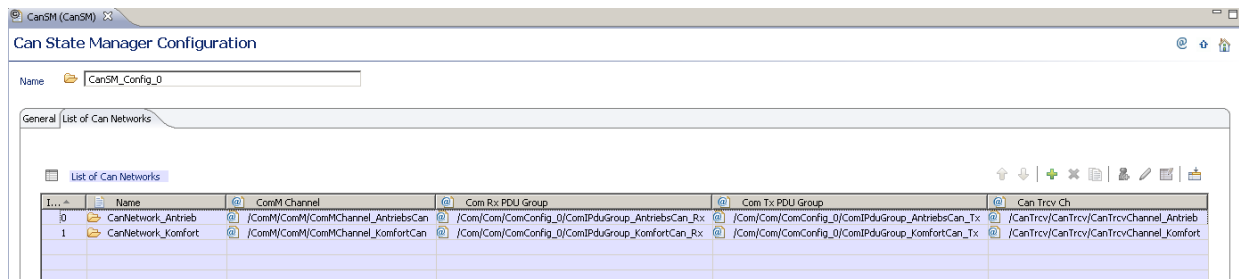


Figure 6.29. CanSM tab List of Nm channels

- You need to configure one network for each connected CAN bus by clicking the **+** button.

For a detailed description of all configuration parameters, refer to the document EB tresos AutoCore module references.

6.3.3.6. Integration notes for FrNm

1. Delayed transition to Bus Sleep when the schedule variant is FRNM_PDU_SCHEDULE_VARIANT_2

This behavior appears only if the following conditions are met:

- The schedule variant is set to `FRNM_PDU_SCHEDULE_VARIANT_2` (NM-Vote and NM-Data in one PDU in the dynamic segment).
- A vote is received in the last FlexRay cycle of the last voting cycle in either one of the repetition cycles.

Because the job list for the dynamic part is executed at the beginning of a FlexRay cycle, the vote for the current FlexRay cycle will be indicated to `FrNm` only in the next FlexRay cycle. In this situation, the `FrNm` state machine will transit to Bus Sleep with a delay of one repetition cycle.

According to FRNM372, the main function should be scheduled only at the beginning of the next cycle if the FlexRay NM vector is only available at the end of a cycle. A vote will be ignored in the last repetition cycle due to the fact that a vote change can only be done at repetition cycle boundary. This behavior will not get visible on the bus at the end of the last repetition cycle, so the ECU transits to Bus Sleep state.

This behavior can be avoided if the main function is scheduled to run between the job list of the current FlexRay cycle and the job list for the next one. This condition implies that the job list must be manually adapted to fit the use case.

6.3.3.7. Integration notes for CanNm

1. CanNm sends default values to PduR if the Com I-PDU group is stopped

This behavior appears only if the following condition is met:

- ▶ The Com I-PDU group corresponding to Tx EIRA is stopped

Because of the stopped I-PDU group, the call to the `PduR_CanNmTriggerTransmit()` API (in the context of `CanNm_GetPduUserData()`) returns `E_NOT_OK`. This leads to the sending of the default value `0xFF` for all EIRA bytes.

This behavior can be avoided if the Com I-PDU group corresponding to Tx EIRA is always started.

6.3.3.8. Integration notes for UdpNm

No integration notes available at the moment.

6.4. Adjacent layer property files

6.4.1. Overview

This chapter explains the concept of adjacent layer property files for EB tresos AutoCore Generic (ACG) modules.

- ▶ [Section 6.4.2, “Background information”](#) explains all aspects of module properties in EB tresos AutoCore Generic.
- ▶ [Section 6.4.3, “Property file”](#) describes the format and content of the property file.
- ▶ [Section 6.4.4, “Use case: Getting the PDU-ID”](#) shows exemplarily how to retrieve PDU-IDs from modules.

6.4.2. Background information

Code generators for EB tresos AutoCore modules (Module Code Generators – MCGs) often need some knowledge of adjacent modules. For example, many of the module API function names do not follow a common

scheme. Some API function names even vary depending on the calling module. On the other hand, MCGs should be implemented in a *generic* way. This means they should be able to get the needed information at run-time, so that new adjacent modules can be introduced after the MCG was implemented.

Thus, modules must publish some properties. They do this in module property files. These property files can only be helpful if there is a common property scheme. This scheme is defined in the following, along with custom XPath functions to retrieve the property values. The XPath functions are implemented in the EB tresos AutoCore Generic 8 Base product.

This section provides information on the following topics:

- ▶ Property format in [Section 6.4.2.1, “Property format”](#)
- ▶ Property getter XPath functions in [Section 6.4.2.2, “Property getter XPath functions”](#)
- ▶ List of properties and functions in [Section 6.4.2.3, “List of properties and functions”](#)

6.4.2.1. Property format

A property consists of a property name and a property value, separated by a colon.

Example: `Com.AdjacentLayerConfig.ApiName.V1.IfRxIndication.PduR:Com_RxIndication`

6.4.2.1.1. Property name

- ▶ The property name consists of particles, separated by periods.
- ▶ A particle must not contain periods, colons, or whitespace.
- ▶ The semantics of the particles depend on their position within the property name.

For the example property above, the particles have the following semantics:

`Com`

The name of the module to which this property belongs. Usually a module's property file only contains its own properties, but there may be exceptions to this rule.

`AdjacentLayerConfig`

A constant string to distinguish the properties of this schema from other properties.

`ApiName`

The semantic of the property.

`V1`

The schema version of the property.

`IfRxIndication.PduR`

Bound parameters, one per particle.

6.4.2.1.2. Property value

- ▶ A property value is either a single value or a list value.
- ▶ A list value is a comma-separated list of scalar values.
- ▶ A scalar value is some string that contains free variables of the form `<xxx>`.

To get property values, EB tresos Studio provides a set of generic functions, e.g.:

- ▶ `ecu:has(name)` to check if a property `<name>` exists
- ▶ `ecu:get(name)` to get the string value of a property
- ▶ `ecu:list(name)` to get an array of string values of a property that contains a comma-separated list

However, these functions do not know the structure of the EB tresos AutoCore module property names. Therefore, it is better to define specialized functions so that the individual MCGs do not have to know this structure.

6.4.2.2. Property getter XPath functions

6.4.2.2.1. Generic signature

To read a property with the particles `<module>.AdjacentLayerConfig.<prop>.V1.<parameters>`, the generic function signature is `asc:get<prop>(module, parameters...)`.

Example

For reading the property `<callee>.AdjacentLayerConfig.ApiName.V1.<name>.<caller>`, the function is `asc:getApiFunctionName(callee, caller, name)` where:

`callee`

is the called module, i. e., the module to which the resulting API function belongs

`caller`

is the calling module

`name`

is the generic function name, in the above example `IfRxIndication`.

Note that in some cases (like in the given example), the name of the function cannot be directly derived from the property name. In the example, the property semantic is `ApiName`, while the function is named `asc:getApi-`

`FunctionName()`. Also, the order of the parameters might be different, or some parameters might be missing in the property name. All this depends on the requirements of the concrete XPath function.

6.4.2.2.2. Property generalization

For many modules, some of the property values could be defined in a generic way.

For example, the property semantic `RelXPathPduId` gives the relative XPath from the PDU to the PDU-ID. The parameters of this property are:

`layer`
the layer type for which the PDUs are used: `ForUpperLayer` or `ForLowerLayer`

`apiType`
the API type: `If` or `Tp`

`direction`
the communication direction: `Rx` or `Tx`

For most modules, this relative path is independent of the parameters. For example, the following eight properties have the same value. You could add them as follows:

```
Module.AdjacentLayerConfig.RelXPathPduId.V1.ForUpperLayer.If.Rx:../PduId
Module.AdjacentLayerConfig.RelXPathPduId.V1.ForUpperLayer.If.Tx:../PduId
Module.AdjacentLayerConfig.RelXPathPduId.V1.ForUpperLayer.Tp.Rx:../PduId
Module.AdjacentLayerConfig.RelXPathPduId.V1.ForUpperLayer.Tp.Tx:../PduId
Module.AdjacentLayerConfig.RelXPathPduId.V1.ForLowerLayer.If.Rx:../PduId
Module.AdjacentLayerConfig.RelXPathPduId.V1.ForLowerLayer.If.Tx:../PduId
Module.AdjacentLayerConfig.RelXPathPduId.V1.ForLowerLayer.Tp.Rx:../PduId
Module.AdjacentLayerConfig.RelXPathPduId.V1.ForLowerLayer.Tp.Tx:../PduId
```

However, you can also omit the parameters and thus add only one property as follows:

```
Module.AdjacentLayerConfig.RelXPathPduId.V1:../PduId
```

The XPath function call `asc:getXPathPduId('ForUpperLayer', 'Module', 'If', 'Rx')` uses the property `Module.AdjacentLayerConfig.RelXPathPduId.V1` if it cannot find the originally searched property.

For example, a function might need the following property:

```
<module>.AdjacentLayerConfig.<semantics>.<version v>.<param 1>.<pa-
ram 2>.<param ...>.<param n-1>.<param n>
```

The function searches for this property according to the following search order:

1. Match for current version v

1.1. Complete property (exact match)

```
<module>.AdjacentLayerConfig.<semantics>.<version v>.<param 1>.<param 2>.<param ...>.<param n-1>.<param n>
```

1.2. Generalization

```
<module>.AdjacentLayerConfig.<semantics>.<version v>.<param 1>.<param 2>.<param ...>.<param n-1>
```

1.3. Further generalization

```
<module>.AdjacentLayerConfig.<semantics>.<version v>.<param 1>.<param 2>.<param ...>
```

1.4. [...]

1.5. Complete generalization

```
<module>.AdjacentLayerConfig.<semantics>.<version v>
```

2. Match for previous version $v-1$

2.1. Complete property (exact match)

```
<module>.AdjacentLayerConfig.<semantics>.<version v-1>.<param 1>.<param 2>.<param ...>.<param n-1>.<param n>
```

2.2. Generalization

```
<module>.AdjacentLayerConfig.<semantics>.<version v-1>.<param 1>.<param 2>.<param ...>.<param n-1>
```

2.3. Further generalization

```
<module>.AdjacentLayerConfig.<semantics>.<version v-1>.<param 1>.<param 2>.<param ...>
```

2.4. [...]

2.5. Complete generalization

```
<module>.AdjacentLayerConfig.<semantics>.<version v-1>
```

3. [..]

4. Match for initial version 1

4.1. Complete property (exact match)

```
<module>.AdjacentLayerConfig.<semantics>.<version 1>.<pa-  
ram 1>.<param 2>.<param ...>.<param n-1>.<param n>
```

4.2. Generalization

```
<module>.AdjacentLayerConfig.<semantics>.<version 1>.<pa-  
ram 1>.<param 2>.<param ...>.<param n-1>
```

4.3. Further generalization

```
<module>.AdjacentLayerConfig.<semantics>.<version 1>.<pa-  
ram 1>.<param 2>.<param ...>
```

4.4. [..]

4.5. Complete generalization

```
<module>.AdjacentLayerConfig.<semantics>.<version 1>
```

5. Use of a default value if defined, or NULL. If applicable, default values are defined implicitly in the XPath function implementation.

6.4.2.2.3. Binding free variables

As stated in [Section 6.4.2.1.2, "Property value"](#), property values may contain free variables of the form `<variable>`. The XPath functions bind these variables to values, i. e., they replace the free variables in the property values with the respective parameter values.

For example, the property `SomeProperty` may have the common structure `<module>.AdjacentLayerConfig.SomeProperty.V1.<layerType>.<apiType>.<direction>`, with the corresponding XPath function `asc:getSomeProperty(module, layerType, apiType, direction)`.

For a module `Module`, the property definition might be:

```
Module.AdjacentLayerConfig.SomeProperty.V1.ForLowerLayer:ModLow_<direction>_<layerType>
```

Any call to `asc:getSomeProperty()` with parameters `module="Module"` and `layerType="ForLowerLayer"` finds the property value `"ModLow_direction_layerType"` with the two free variables `direction` and `layerType`, provided that there is no other specialized property whose name also contains particles for `apiType` and `direction` that match the respective function arguments.

Note that the function argument `apiType` is not a free variable, although it neither is part of the property name. On the other hand, `layerType` is a free variable, although it is already fixed in the property name. All of this is perfectly valid.

The XPath function then binds the free variables to the function parameters. The possible results of the function call in this example are:

```
asc:getSomeProperty("Module", "ForLowerLayer", "If", "Rx")
    "ModLow_Rx_ForLowerLayer"

asc:getSomeProperty("Module", "ForLowerLayer", "If", "Tx")
    "ModLow_Tx_ForLowerLayer"

asc:getSomeProperty("Module", "ForLowerLayer", "Tp", "Rx")
    "ModLow_Rx_ForLowerLayer"

asc:getSomeProperty("Module", "ForLowerLayer", "Tp", "Tx")
    "ModLow_Tx_ForLowerLayer"
```

6.4.2.3. List of properties and functions

This table provides the data types that are used in the following sections.

Data type	Description
String	An arbitrary text string.
Integer	An integer number.
Identifier	A valid C identifier.
Module	The name of a module (a string) to which a called function belongs.
HandleIdPolicy	A string representing the handle id policy. Allowed values are currently AUTOSAR32 and AUTOSAR40 .
XPath	A string constituting a valid XPath expression.
ApiType	The type of API. This is a string that is restricted to the values If and Tp.
ApiFeature	The type of API features that can be enabled or disabled. This is a string that is restricted to be a valid C identifier.
ByteOrdering	A string representing a Byte ordering policy. Allowed values are LE and BE.
Layer	A string representing the relative layer that a property is meant for. Allowed values are ForUpperLayer and ForLowerLayer.
Direction	A string representing the communication direction. Allowed values are Rx and Tx.
SupportType	A string representing the type of support for a feature. Allowed values are Mandatory, Optional and Unsupported.
x	A list of values of type x.

Data type	Description
<code>Maybe x = Nothing Just x</code>	A type with an optional value of type <code>x</code> , but without a default value. A property with type <code>Maybe x</code> can either have a value of type <code>x</code> (denoted as <code>Just x</code>) or none (<code>Nothing</code>).

Property names may contain variable parts (called *parameters* or *bound parameters* in the description above). Properties can therefore be seen as functions which map the set of particles in the variable part to a property value.

In the following table, the properties are described like functions. The signature column shows the types of the parameters and the corresponding property value. The format of the signature description is borrowed from the Haskell programming language. Other functional languages use similar syntax.

The following table shows the mandatory and optional properties for the `AdjacentLayerConfig` feature.

Mandatory properties

Signature	Description
<code>handleIDPolicyVersion :: Module -> HandleIdPolicy</code>	Name schema <code><module>.AdjacentLayerConfig.HandleIDPolicyVersion</code>
<code>moduleConfigName :: Module -> String</code>	Name schema <code><module>.AdjacentLayerConfig.ModuleConfigName</code>

Optional properties

Signature	Description
<code>absXPathPduRef :: Module -> Layer -> ApiType -> Maybe XPath</code>	The XPath to find all PDU EcuC reference containers of a module. In addition to the variables in the name schema, the property value may contain the free variable <code><configIndex></code> . Name schema <code><module>.AdjacentLayerConfig.AbsXPathPduRef.V1.<layer>.<apiType>.<direction></code>
<code>apiName :: Module -> Identifier -> Module -> Identifier</code>	A name of an API function provided by the callee module to the caller module. Name schema <code><callee>.AdjacentLayerConfig.ApiName.V1.<name>.<caller></code> Default <code>"<callee>_<caller><name>"</code>

Signature	Description
<pre>includeFileNames :: Module -> ApiType -> Layer -> Module -> [Filename]</pre>	<p>A list of include files that adjacent layer modules need in order to use the module's API.</p> <p>Name schema</p> <pre><callee>.AdjacentLayerConfig.IncludeFile- Names.V1.<apiType>.<layer>.<caller></pre> <p>Default</p> <pre>[]</pre>
<pre>interfaceSupport :: Module -> ApiType -> SupportType</pre>	<p>Information on whether a given API type (If/Tp) is (optionally) supported.</p> <p>Name schema</p> <pre><module>.AdjacentLayerConfig.InterfaceSup- port.V1.<apiType></pre> <p>Default</p> <pre>"Unsupported"</pre>
<pre>layerSupport :: Module -> Layer -> SupportType</pre>	<p>Information on whether a module (optionally) provides services for an upper or lower layer.</p> <p>Name schema</p> <pre><module>.AdjacentLayerConfig.LayerSup- port.V1.<layer></pre> <p>Default</p> <pre>"Unsupported"</pre>
<pre>providedApiFeature :: Module -> ApiFeature -> Sup- portType</pre>	<p>Information on whether a module (optionally) provides a given API feature to its adjacent layer modules.</p> <p>Name schema</p> <pre><module>.AdjacentLayerConfig.ProvidedApiFea- ture.V1.<layer>.<apiFeature></pre> <p>Default</p> <pre>"Unsupported"</pre>
<pre>relXPathPduId :: Module -> Layer -> ApiType -> Maybe XPath</pre>	<p>The relative XPath from a PDU EcuC reference as given by absXPathPduRef to its handle ID.</p> <p>Name schema</p> <pre><module>.AdjacentLayerConfig.RelXPathP- duId.V1.<layer>.<apiType>.<direction></pre>

Signature	Description
<code>requiredApiFeature :: Module -> ApiFeature -> SupportType</code>	<p>Information on whether a module (optionally) requires a given API feature from its adjacent layer modules.</p> <p>Name schema</p> <pre><module>.AdjacentLayerConfig.RequiredApiFeature.V1.<layer>.<apiFeature></pre> <p>Default</p> <pre>"Unsupported"</pre>
<code>supportedAdjacentModules :: Module -> Layer -> [Module]</code>	<p>A list of modules that are supported by the module as adjacent layer modules, depending on the layer type. An empty list means that all modules are supported. If <code>LayerSupport(<module>, <layer>)</code> is "Unsupported", then this property has no meaning.</p> <p>Name schema</p> <pre><module>.AdjacentLayerConfig.SupportedAdjacentModules.V1.<layer></pre> <p>Default</p> <pre>[]</pre>

The following table describes the mandatory and optional properties that are used by the above defined functions, but which are not part of the `AdjacentLayerConfig` feature.

Mandatory properties

Signature	Description
<code>endianness :: ByteOrdering</code>	<p>The endianness of the target platform.</p> <p>Name schema</p> <pre>Cpu.Byteorder</pre>

Optional properties

Signature	Description
<code>alignment :: Identifier -> Maybe Integer</code>	<p>The alignment requirement in Bytes of a primitive or derived data type.</p> <p>Name schema</p> <pre>Basetypes.<name>.Alignment</pre> <pre>Derivedtypes.<name>.Alignment</pre>
<code>arrayAlignment :: Maybe Integer</code>	<p>The alignment requirement in Bytes for an array type.</p> <p>Name schema</p> <pre>Complextypes.array.Alignment</pre>

Signature	Description
<code>mapping :: Identifier -> Maybe Identifier</code>	<p>The mapping of a data type to another. This correlates to a <code>typedef</code> in the C programming language.</p> <p>Name schema</p> <p><code>Basetypes.<name>.Mapping</code></p> <p><code>Derivedtypes.<name>.Mapping</code></p>
<code>size :: Identifier -> Maybe Integer</code>	<p>The Byte size of a primitive or derived data type.</p> <p>Name schema</p> <p><code>Basetypes.<name>.Size</code></p> <p><code>Derivedtypes.<name>.Size</code></p>
<code>structAlignment :: Maybe Integer</code>	<p>The alignment requirement in Bytes for a struct type.</p> <p>Name schema</p> <p><code>Complextypes.struct.Alignment</code></p>

The following table shows the XPath functions that are implemented in namespace `asc`. They are implemented in the EB tresos AutoCore Generic 8 Base module.

Signature functions

Function	Description
<code>getConfigSignature(nodes) : integer</code>	Get the signature of a given set of nodes. The nodes are first sorted according to their paths, i.e., regardless of the order of the original list. Afterwards, the signature is calculated as a hash value of the values of the nodes.
<code>getPlatformSignature() : integer</code>	Calculate the platform signature from all basic data types (see the platform signature description).

Platform property functions

Function	Description
<code>getArrayAlignment() : integer</code>	<p>Get the alignment requirements for arrays.</p> <p>Properties</p> <p><code>arrayAlignment</code></p>
<code>getDataTypeAlignment(name) : integer</code>	Get the alignment requirement in Bytes for a given basic or derived data type. The function follows the data type mappings until it finds a type that is not mapped to another. The value of the alignment property for this most primitive type is returned.

Function	Description
	Properties mapping alignment
<code>getDataTypeSize(name): integer</code>	<p>Get the Byte size of a given basic or derived data type. The function follows the data type mappings until it finds a type that is not mapped to another. The value of the size property for this most primitive type is returned.</p> Properties mapping size
<code>getEndianness(): string</code>	<p>Get the platform CPU endianness</p> Properties endianness
<code>getStructAlignment(): integer</code>	<p>Get the alignment requirement structs.</p> Properties structAlignment

Module property functions

Function	Description
<code>getApiFunctionName(caller, callee, name): string</code>	<p>Get the name of an API function.</p> Properties apiName Default <callee>_<caller><name>
<code>getHandleIDPolicyVersion(module): string</code>	<p>Get the version of the supported handle ID policy.</p> Properties handleIDPolicyVersion
<code>getIncludes(caller, callee, layer, apiType): list of strings</code>	<p>Get a list of include files.</p> Properties includeFileNames Defaults <layer> == "ForUpperLayer" # [<callee>_<caller>.h]

Function	Description
	<pre><layer> == "ForLowerLayer" # [<callee>_- <caller>_Cbk.h]</pre>
<pre>getInterfaceSupport (module, apiType): string</pre>	<p>Retrieves the info whether a module given by its name provides a Tp/If interface.</p> <p>Properties</p> <p>layerServiceSupport</p>
<pre>getLayerServiceSupport (module, layer, apiType): string</pre>	<p>Retrieves the info whether a module given by its name provides a Tp/If interface for upper/lower layer modules.</p> <p>Properties</p> <p>layerServiceSupport</p>
<pre>getLayerSupport (module, layer): string</pre>	<p>Retrieves the info whether a module given by its name can/must serve as an upper layer to any module.</p> <p>Properties</p> <p>layerServiceSupport</p>
<pre>getPduId (module, layer, apiType, direction, pduRef): integer</pre>	<p>Get the PDU-ID for a PDU reference.</p> <p>Properties</p> <p>relXPathPduId</p>
<pre>getPdus (module, layer, apiType, direction, configIndex, pduRef): list of nodes</pre>	<p>Get all PDU reference nodes that reference the same EcuC PDU as a given reference node.</p> <p>Properties</p> <p>absXPathPduRef</p>
<pre>getProvidedApiFeatureSupport (module, layer, apiFeature): string</pre>	<p>Check whether a given API feature is provided by a given module.</p> <p>Properties</p> <p>providedApiFeature</p>
<pre>getRequiredApiFeatureSupport (module, layer, apiFeature): string</pre>	<p>Check whether a given API feature is required by a given module.</p> <p>Properties</p> <p>requiredApiFeature</p>
<pre>getXPathPduId (layer, module, apiType, direction): string</pre>	<p>Get the relative XPath from a node representing a PDU reference to its PduId.</p> <p>Properties</p> <p>relXPathPduId</p>

Function	Description
<code>getXPathPduRef(layer, module, apiType, direction, configIndex): string</code>	<p>Get an XPath for finding all PDU references that a module given by its name provides as upper or lower layer via Tp or If for reception/transmission.</p> <p>Properties</p> <p><code>absXPathPduRef</code></p>
<code>isPduIdValid(module, layer, apiType, direction, pduRef): boolean</code>	<p>Check whether the PDU-ID for a PDU is existing, enabled, and a valid, non-negative integer number.</p> <p>Properties</p> <p><code>relXPathPduId</code></p>
<code>isPropertyFileValid (module, caller): boolean</code>	<p>Check whether the mandatory properties for a given module and caller pair are existing.</p> <p>Properties</p> <p><code>moduleConfigName</code></p>

6.4.3. Property file

A property file may contain blank lines, comments starting with #, and properties.

6.4.3.1. Property file template

The following shows a template for the `AdjacentLayer.properties` file.

```
# AdjacentLayer.properties file of module <Modulename>.
#
# The module name corresponds to the name of attribute 'categoryType' in the
# plugin.xml of the adjacent module and must be identical to the PduRBswModules
# Container-short name.
#
# The properties in 'Module Type Dependent Data' must be provided in dependence on
# particles ForUpperLayer|ForLowerLayer, If|Tp ('If' for handling nonTP-PDUs, 'Tp'
# for handling TP-PDUs) and Rx/Tx. If it's not provided it's considered as
# 'Unsupported'.
#
# 'Mandatory', 'Unsupported' and 'Optional' are considered as possible values with
# the following meaning:
# - Mandatory: module supports this API, API shall be used by adjacent layer module
# - Optional: module supports this API (depending on the configuration), API can be
```



```
# used by adjacent layer module
# - Unsupported: module doesn't support this API, API shall not be used by adjacent
# layer module
# The XPath expressions to the references of the relevant PDUs or PDUIDs shall be
# restricted by conditions if possible.
#
# The properties for the 'API Names' are optional:
# - if no value is provided, default name is assigned (<Callee>_<Caller><ApiName>)
# where <ApiName> is the name of the API with API type (e.g. IfRxindication,
# TpRxindication,...) and
# - those APIs provided by the adjacent layer module shall start with '<Callee>_'.
#
# The following lists all property names available.
#
# To design a suitable AdjacentLayer.properties file out of this template,
# the '<Modulename>' shall be replaced by the name of the module throughout the file.
# For those property names needed, the respective values shall be set if different to
# the default value.
# Property names not needed may be removed (e.g. for a Tp module like CanTp
# remove all unnecessary property names related to If).
# '<AdjacentLayerName>' shall be replaced by the name of the adjacent layer name if
# adjacent layer's API names shall be overruled.
# Store the template file as 'AdjacentLayer.properties' in the <Modulename>'s
# resources folder.

# -----
# ----- General Data -----
# -----

# Name of module configuration (i.e name which can be used with
# as:modconf('<ModuleConfigName>'))
<Modulename>.AdjacentLayerConfig.ModuleConfigName:

# Supported AUTOSAR Handle ID Version (allowed values: AUTOSAR32, AUTOSAR40)
<Modulename>.AdjacentLayerConfig.HandleIDPolicyVersion:

# Supported AUTOSAR TP API Version (allowed values: AUTOSAR32, AUTOSAR40)
# only required for PduR upper layer Tp modules
<Modulename>.AdjacentLayerConfig.TPAPIVersion:AUTOSAR40

# Name of include file(s)
# Multiple header files are separated by comma.
# If no header file is provided no property value shall be configured.
<Modulename>.AdjacentLayerConfig.IncludeFileNames.V1.If.ForUpperLayer:
<Modulename>.AdjacentLayerConfig.IncludeFileNames.V1.If.ForLowerLayer:
<Modulename>.AdjacentLayerConfig.IncludeFileNames.V1.Tp.ForUpperLayer:
```




```

<Modulename>.AdjacentLayerConfig.IncludeFileNames.V1.Tp.ForLowerLayer:

# Information which services (If/Tp) a module (optionally) provides for an upper
# or lower layer.
# (allowed values: Mandatory, Unsupported, Optional, default: Unsupported)
<Modulename>.AdjacentLayerConfig.LayerServiceSupport.V1.If.ForUpperLayer:
<Modulename>.AdjacentLayerConfig.LayerServiceSupport.V1.If.ForLowerLayer:
<Modulename>.AdjacentLayerConfig.LayerServiceSupport.V1.Tp.ForUpperLayer:
<Modulename>.AdjacentLayerConfig.LayerServiceSupport.V1.Tp.ForLowerLayer:

# List of modules which are possible as upper or lower layer modules
# (comma separated list or use empty property if module has a generic layer support)
# List is only valid if respective LayerServiceSupport exists.
<Modulename>.AdjacentLayerConfig.LayerSupportList.V1.ForUpperLayer:
<Modulename>.AdjacentLayerConfig.LayerSupportList.V1.ForLowerLayer:

# -----
# -----  Module Type Dependent Data  -----
# -----

# Information on whether a module (optionally) provides a given API feature to its
# adjacent layer modules.
# (allowed values: Mandatory, Unsupported, Optional, default: Unsupported)
# <apiFeature> is the name of the API together with the API type e.g. TpRxIndication
# or IfTriggerTransmit
# <Modulename>.AdjacentLayerConfig.ProvidedApiFeature.V1.<layerType>.<apiFeature>
<Modulename>.AdjacentLayerConfig.ProvidedApiFeature.V1.ForLowerLayer.IfTxConfirmation:
<Modulename>.AdjacentLayerConfig.ProvidedApiFeature.V1.ForLowerLayer.IfTriggerTransmit:
<Modulename>.AdjacentLayerConfig.ProvidedApiFeature.V1.ForUpperLayer.IfCancelTransmit:
<Modulename>.AdjacentLayerConfig.ProvidedApiFeature.V1.ForUpperLayer.TpChangeParameter:
<Modulename>.AdjacentLayerConfig.ProvidedApiFeature.V1.ForUpperLayer.TpCancelReceive:
<Modulename>.AdjacentLayerConfig.ProvidedApiFeature.V1.ForUpperLayer.TpCancelTransmit:

# Information on whether a module (optionally) requires a given API feature from its
# adjacent layer modules.
# (allowed values: Mandatory, Unsupported, Optional, default: Unsupported)
# <apiFeature> is the name of the API together with the API type e.g. TpRxIndication
# or IfTriggerTransmit
# <Modulename>.AdjacentLayerConfig.RequiredApiFeature.V1.<layerType>.<apiFeature>
<Modulename>.AdjacentLayerConfig.RequiredApiFeature.V1.ForLowerLayer.IfCancelTransmit:
<Modulename>.AdjacentLayerConfig.RequiredApiFeature.V1.ForLowerLayer.TpChangeParameter:
<Modulename>.AdjacentLayerConfig.RequiredApiFeature.V1.ForLowerLayer.TpCancelReceive:
<Modulename>.AdjacentLayerConfig.RequiredApiFeature.V1.ForLowerLayer.TpCancelTransmit:
<Modulename>.AdjacentLayerConfig.RequiredApiFeature.V1.ForUpperLayer.IfTxConfirmation:
<Modulename>.AdjacentLayerConfig.RequiredApiFeature.V1.ForUpperLayer.IfTriggerTransmit:

# ----- Data to referenced global Rx I-PDUs (ForLowerLayer, If-module) -----

```



```
# XPath-expression to all references of the relevant Rx I-PDUs
<Modulename>.AdjacentLayerConfig.AbsXPathPduRef.V1.ForLowerLayer.If.Rx:

# relative XPath-expression to associated PduId of above referenced Rx I-PDU
<Modulename>.AdjacentLayerConfig.RelXPathPduId.V1.ForLowerLayer.If.Rx:

# ----- Data to referenced global Tx I-PDUs (ForLowerLayer, If-module) -----
# XPath-expression to all references of the relevant Tx I-PDUs
<Modulename>.AdjacentLayerConfig.AbsXPathPduRef.V1.ForLowerLayer.If.Tx:

# relative XPath-expression to associated PduId of above referenced Tx I-PDU
<Modulename>.AdjacentLayerConfig.RelXPathPduId.V1.ForLowerLayer.If.Tx:

# ----- Data to referenced global Rx I-PDUs (ForLowerLayer, Tp-module) -----
# XPath-expression to all references of the relevant Rx I-PDUs
<Modulename>.AdjacentLayerConfig.AbsXPathPduRef.V1.ForLowerLayer.Tp.Rx:

# relative XPath-expression to associated PduId of above referenced Rx I-PDU
<Modulename>.AdjacentLayerConfig.RelXPathPduId.V1.ForLowerLayer.Tp.Rx:

# ----- Data to referenced global Tx I-PDUs (ForLowerLayer, Tp-module) -----
# XPath-expression to all references of the relevant Tx I-PDUs
<Modulename>.AdjacentLayerConfig.AbsXPathPduRef.V1.ForLowerLayer.Tp.Tx:

# relative XPath-expression to associated PduId of above referenced Tx I-PDU
<Modulename>.AdjacentLayerConfig.RelXPathPduId.V1.ForLowerLayer.Tp.Tx:

# ----- Data to referenced global Rx I-PDUs (ForUpperLayer, If-module) -----
# XPath-expression to all references of the relevant Rx I-PDUs
<Modulename>.AdjacentLayerConfig.AbsXPathPduRef.V1.ForUpperLayer.If.Rx:

# relative XPath-expression to associated PduId of above referenced Rx I-PDU
<Modulename>.AdjacentLayerConfig.RelXPathPduId.V1.ForUpperLayer.If.Rx:

# ----- Data to referenced global Tx I-PDUs (ForUpperLayer, If-module) -----
# XPath-expression to all references of the relevant Tx I-PDUs
<Modulename>.AdjacentLayerConfig.AbsXPathPduRef.V1.ForUpperLayer.If.Tx:

# relative XPath-expression to associated PduId of above referenced Tx I-PDU
<Modulename>.AdjacentLayerConfig.RelXPathPduId.V1.ForUpperLayer.If.Tx:

# ----- Data to referenced global Rx I-PDUs (ForUpperLayer, Tp-module) -----
# XPath-expression to all references of the relevant Rx I-PDUs
<Modulename>.AdjacentLayerConfig.AbsXPathPduRef.V1.ForUpperLayer.Tp.Rx:
```



```
# relative XPath-expression to associated PduId of above referenced Rx I-PDU
<Modulename>.AdjacentLayerConfig.RelXPathPduId.V1.ForUpperLayer.Tp.Rx:

# ----- Data to referenced global Tx I-PDUs (ForUpperLayer, Tp-module) -----
# XPath-expression to all references of the relevant Tx I-PDUs
<Modulename>.AdjacentLayerConfig.AbsXPathPduRef.V1.ForUpperLayer.Tp.Tx:

# relative XPath-expression to associated PduId of above referenced Tx I-PDU
<Modulename>.AdjacentLayerConfig.RelXPathPduId.V1.ForUpperLayer.Tp.Tx:

# The module gateways solely Rx I-PDUs of fixed length when using single or
# FIFO-buffers
# - required at least for modules which use the PduR as upper layer module
# - allowed values: Mandatory, Unsupported, Optional
<Modulename>.AdjacentLayerConfig.StaticPduLength.V1.ForUpperLayer.If.Rx:

# Boolean value to associated data provision of above referenced Tx I-PDU with respect
# to 'Direct' data provision. This is required for consistency check of PduR routing
# path Tx buffer configuration and lower layer data provision.
# - required at least for modules which use the PduR as upper layer module
# Either set to:
# - true(), e.g. CanIf (solely direct data provision is provided for I-PDUs)
# - false(), e.g. LinIf (solely trigger transmit data provision is provided for I-PDUs)
# - relative XPath-expression delivering true() and false() in dependence on above
#   referenced Tx I-PDUs, e.g. FrIf with 'node:value(..FrIfImmediate)'
#   (direct and trigger transmit data provision can be provided for I-PDUs)
<Modulename>.AdjacentLayerConfig.DataProvisionDirect.V1.ForUpperLayer.If.Tx:

# ----- Data for Retransmission (ForUpperLayer, Tp-module) -----
# Support of the Retransmission Feature
# - required at least for modules which use the PduR as upper layer module
# - allowed values: Mandatory, Unsupported, Optional)
<Modulename>.AdjacentLayerConfig.Retransmission.V1.ForUpperLayer.Tp:

# -----
# ----- API Names -----
# -----

# API functions are provided by the callee module to the caller module with
# default pattern <callee>_<caller><name> where <name> reflects the
# particle of property <caller>.AdjacentLayerConfig.ApiName.V1.<name>
<Modulename>.AdjacentLayerConfig.ApiName.V1.IfRxIndication:
<Modulename>.AdjacentLayerConfig.ApiName.V1.IfTxConfirmation:
<Modulename>.AdjacentLayerConfig.ApiName.V1.IfTriggerTransmit:
<Modulename>.AdjacentLayerConfig.ApiName.V1.TpStartOfReception:
<Modulename>.AdjacentLayerConfig.ApiName.V1.TpCopyRxData:
```

```

<ModuleName>.AdjacentLayerConfig.ApiName.V1.TpRxIndication:
<ModuleName>.AdjacentLayerConfig.ApiName.V1.TpCopyTxData:
<ModuleName>.AdjacentLayerConfig.ApiName.V1.TpTxConfirmation:
<ModuleName>.AdjacentLayerConfig.ApiName.V1.IfTransmit:
<ModuleName>.AdjacentLayerConfig.ApiName.V1.IfCancelTransmit:
<ModuleName>.AdjacentLayerConfig.ApiName.V1.TpTransmit:
<ModuleName>.AdjacentLayerConfig.ApiName.V1.TpChangeParameter:
<ModuleName>.AdjacentLayerConfig.ApiName.V1.TpCancelReceive:
<ModuleName>.AdjacentLayerConfig.ApiName.V1.TpCancelTransmit:

# Overrule API names provided by the adjacent layer's <AdjacentLayerName> property
# file, e.g. to overrule the default API name PduR_ComIfTransmit of the PduR by
# PduR_ComTransmit of the Com properties file.
# PduR.AdjacentLayerConfig.ApiName.V1.IfTransmit.Com:PduR_ComTransmit
<AdjacentLayerName>.AdjacentLayerConfig.ApiName.V1.IfRxIndication.<ModuleName>:
<AdjacentLayerName>.AdjacentLayerConfig.ApiName.V1.IfTxConfirmation.<ModuleName>:
<AdjacentLayerName>.AdjacentLayerConfig.ApiName.V1.IfTriggerTransmit.<ModuleName>:
<AdjacentLayerName>.AdjacentLayerConfig.ApiName.V1.TpStartOfReception.<ModuleName>:
<AdjacentLayerName>.AdjacentLayerConfig.ApiName.V1.TpCopyRxData.<ModuleName>:
<AdjacentLayerName>.AdjacentLayerConfig.ApiName.V1.TpRxIndication.<ModuleName>:
<AdjacentLayerName>.AdjacentLayerConfig.ApiName.V1.TpCopyTxData.<ModuleName>:
<AdjacentLayerName>.AdjacentLayerConfig.ApiName.V1.TpTxConfirmation.<ModuleName>:
<AdjacentLayerName>.AdjacentLayerConfig.ApiName.V1.IfTransmit.<ModuleName>:
<AdjacentLayerName>.AdjacentLayerConfig.ApiName.V1.IfCancelTransmit.<ModuleName>:
<AdjacentLayerName>.AdjacentLayerConfig.ApiName.V1.TpTransmit.<ModuleName>:
<AdjacentLayerName>.AdjacentLayerConfig.ApiName.V1.TpChangeParameter.<ModuleName>:
<AdjacentLayerName>.AdjacentLayerConfig.ApiName.V1.TpCancelReceive.<ModuleName>:
<AdjacentLayerName>.AdjacentLayerConfig.ApiName.V1.TpCancelTransmit.<ModuleName>:

```

6.4.4. Use case: Getting the PDU-ID

6.4.4.1. Getting the PDU-ID from own module

The following example shows how to retrieve the PDU-ID of a Tx PDU for the `CanIf` module. The example assumes that `node:current()` is the `EcuC` reference parameter of a Tx I-PDU for which the PDU-ID shall be read.

Note: For the `layer` parameter, the value `ForUpperLayer` is used. This is because the PDU-ID is also provided for the `PduR` module, which is the upper layer of `CanIf`.



Getting the PDU-ID from own module

Step 1

Check if the PDU-ID is valid:

```
asc:isPduIdValid('CanIf', 'ForUpperLayer', 'If', 'Tx', node:current())
```

Step 2

If the PDU-ID is valid, get the value of the PDU-ID:

```
asc:getPduId('CanIf', 'ForUpperLayer', 'If', 'Tx', node:current())
```

6.4.4.2. Getting the PDU-ID from adjacent module

The following example shows how to get the PDU-ID for a Tx I-PDU from `PduR` for a `<Bus>If` module. `PduR` is the upper layer of `<Bus>If`. Therefore, in the XPath functions which use properties from `PduR`, the value `ForLowerLayer` shall be used for the `layer` parameter. The example assumes that `node:current()` is the `EcuC` reference parameter of a Tx I-PDU in the `<Bus>If` module for which the PDU-ID shall be read.



Getting the PDU-ID from adjacent module

Step 1

Get the number of PDUs which are referenced by the adjacent layer, here `PduR`:

```
count(asc:getPdus('PduR', 'ForLowerLayer', 'If', 'Tx', 1, node:current()))
```

Step 2

If XPath returned a match, check if the PDU-ID is valid:

```
asc:isPduIdValid('PduR', 'ForLowerLayer', 'If', 'Tx', asc:getPdus('PduR', 'ForLowerLayer', 'If', 'Tx', 1, node:current())[1])
```

Note: `asc:getPdus` returns a list of nodes. To get the first node, use `[1]`.

Step 3

If the PDU-ID is valid, get the value of the PDU-ID:

```
asc:getPduId('PduR', 'ForLowerLayer', 'If', 'Tx', asc:getPdus('PduR', 'ForLowerLayer', 'If', 'Tx', 1, node:current())[1])
```

6.5. Post-build support

6.5.1. Overview

This chapter explains the post-build concept as defined by AUTOSAR and describes the support in EB tresos AutoCore and EB tresos Studio. The post-build concept supports the use-case in which you change the initial data in a subset of configuration parameters after you have generated, built and downloaded the software to an ECU.

- ▶ [Section 6.5.2, “Background information”](#) explains the post-build concepts in AUTOSAR, EB tresos AutoCore and EB tresos Studio.
- ▶ [Section 6.5.3, “Using post-build support in EB tresos AutoCore”](#) describes a workflow that you can use to learn how to use the post-build feature in EB tresos AutoCore.

6.5.2. Background information

The section provides background information on the following topics:

- ▶ The post-build concept in AUTOSAR [Section 6.5.2.1, “Post-build concept in AUTOSAR”](#)
- ▶ The post-build concept in EB tresos AutoCore [Section 6.5.2.2, “Post-build support in EB tresos AutoCore”](#)
- ▶ The post-build concept in EB tresos Studio [Section 6.5.2.3, “Post-build support in EB tresos Studio”](#)

6.5.2.1. Post-build concept in AUTOSAR

This section provides a brief overview of the post-build concept as defined by AUTOSAR. You find detailed information in the AUTOSAR document *Specification of ECU Configuration*.

In AUTOSAR two kinds of post-build configuration are defined:

- ▶ post-build loadable
- ▶ post-build selectable

6.5.2.1.1. The post-build loadable concept

The concept of post-build loadable describes the use-case where parts of the configuration data that have been flashed into an ECU can be updated without having to create the complete binary again. For the configuration data to be loadable, AUTOSAR specifies that the data must be stored at fixed locations in the ECU.

[Figure 6.30, “The post-build loadable concept”](#) depicts the post-build loadable concept in AUTOSAR.

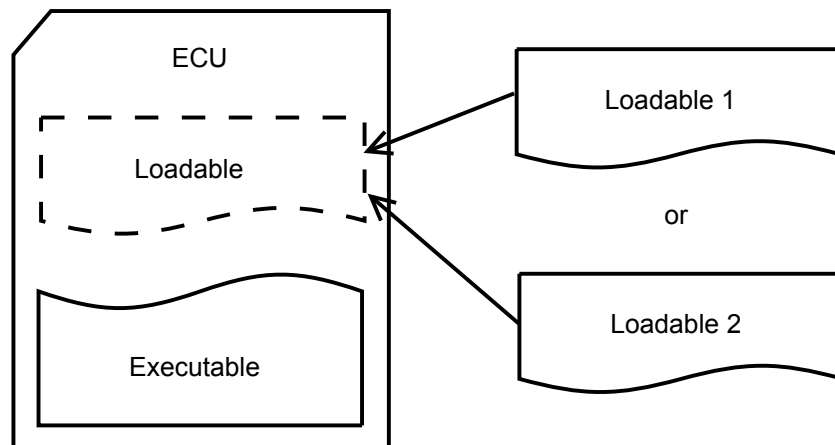


Figure 6.30. The post-build loadable concept

6.5.2.1.2. The post-build selectable concept

The concept of post-build selectable means that several sets of configuration data are created and loaded into the ECU at the same time. On start-up, one of these configurations is selected. Note that a pure post-build selectable solution leads to multiple copies of the configuration data in the ECU. This is especially inefficient if only a few configuration parameters are different.

[Figure 6.31, “The post-build selectable concept”](#) depicts the post-build selectable concept in AUTOSAR.

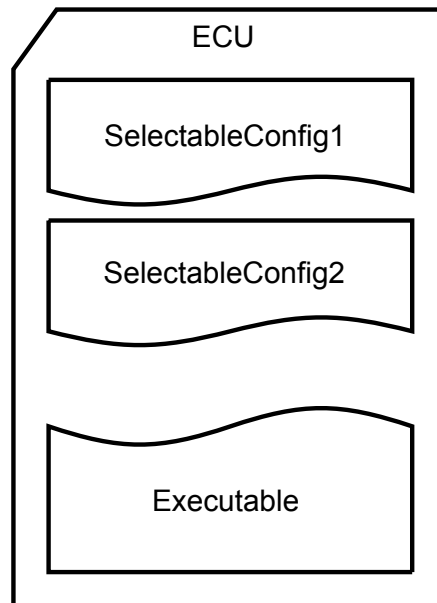


Figure 6.31. The post-build selectable concept

6.5.2.2. Post-build support in EB tresos AutoCore

This section enables you to understand the basic concepts of the post-build support as it is implemented in EB tresos AutoCore. The concepts are described in the following sections starting with an overview and followed by a more detailed description later on.

- ▶ [Section 6.5.2.2.1, “Overview of the post-build loadable concept”](#) gives you an overview of the post-build loadable concept together with a typical use-case.
- ▶ [Section 6.5.2.2.2, “Post-build selectable concept”](#) explains the post-build selectable concept in EB tresos AutoCore.
- ▶ [Section 6.5.2.2.3, “Side allocation support”](#) explains the support for side allocation in EB tresos AutoCore.
- ▶ [Section 6.5.2.2.4, “Relocatable configuration data in the post-build concept”](#) explains the concept of relocatable configuration data in the post-build concept in EB tresos AutoCore.
- ▶ [Section 6.5.2.2.5, “Binary code generation in the post-build concept”](#) explains the concept of binary generation in the post-build concept in EB tresos AutoCore.
- ▶ [Section 6.5.2.2.6, “The post-build configuration manager module `PbcfgM`”](#) gives an overview of the EB module that is used to manage the post-build concept in EB tresos AutoCore.
- ▶ [Section 6.5.2.2.7, “Post-build support in individual modules”](#) describes the implementation of the post-build concept in the basic software (BSW) modules.

If you are already familiar with the basic concepts of post-build support within EB tresos AutoCore, you may want to skip this chapter and proceed to the instruction chapters in [Section 6.5.3, “Using post-build support in EB tresos AutoCore”](#).

6.5.2.2.1. Overview of the post-build loadable concept

The concept of the post-build loadable functionality allows you to change the configuration of a basic software module without the need to rebuild the whole ECU software. A post-build configuration is generated that can be downloaded directly to the ECU.

The following steps describe a typical use-case that uses the post-build loadable concept using EB tresos Studio and EB tresos AutoCore.

1. Configuration of all ECU parameters

First, all ECU configuration parameters are configured. Based on this configuration, EB tresos Studio generates ECU code together with precompile time and link time configuration data (pre-build time data) which is linked to the application. EB tresos Studio also generates post-build configuration data which is written to a separate memory location that you can update later. The generated software and data is part of the ECU's software and is downloaded to the ECU.

[Figure 6.32, “The post-build loadable use-case step 1”](#) depicts the first step in the post-build use-case.

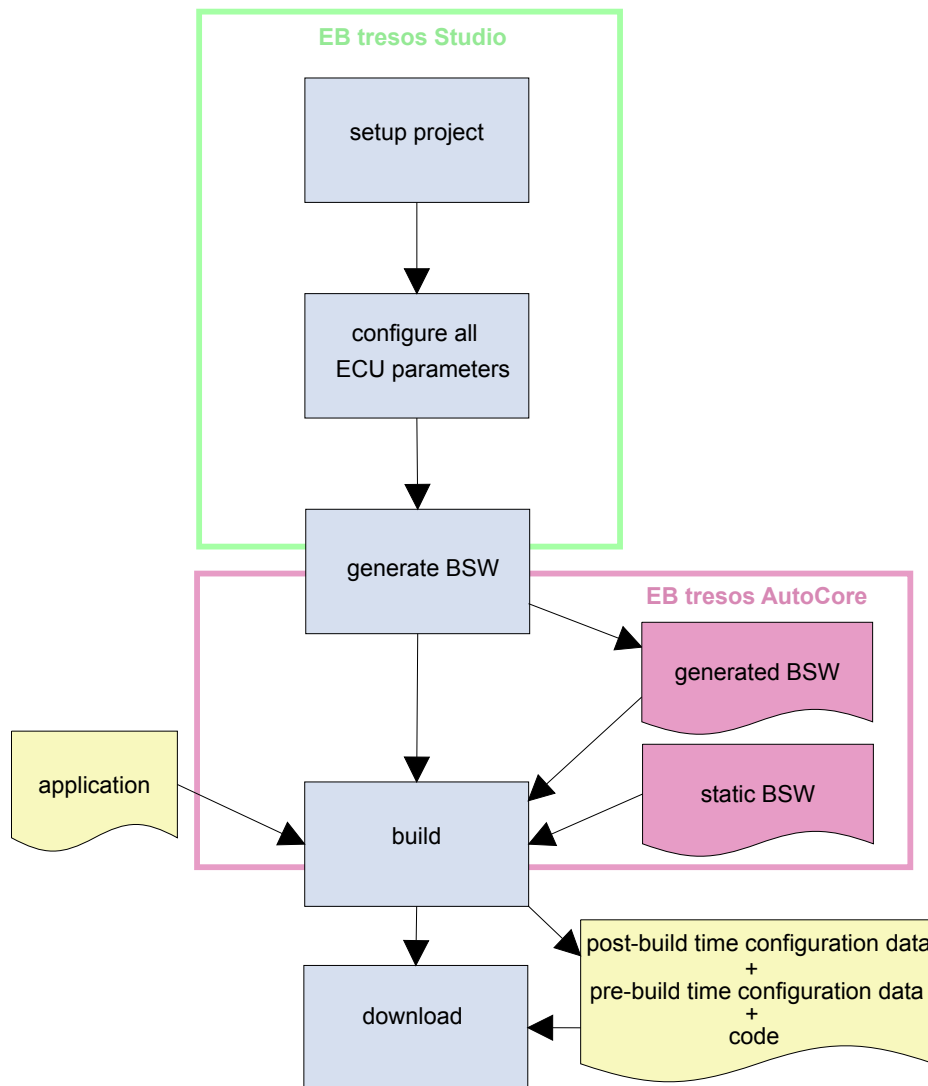


Figure 6.32. The post-build loadable use-case step 1

2. Change of post-build configuration parameters

In a second step, configuration parameters of configuration class `PostBuild` can be updated in EB tresos Studio. During code generation, the post-build configuration data is updated without any need to change the ECU's software or the precompile time or link time configuration data (pre-build time data). The generated post-build configuration data can then be downloaded to the ECU without affecting the ECU's software or pre-build time data configuration.

[Figure 6.33, “The post-build loadable use-case step 2”](#) depicts the second step in the post-build use-case.

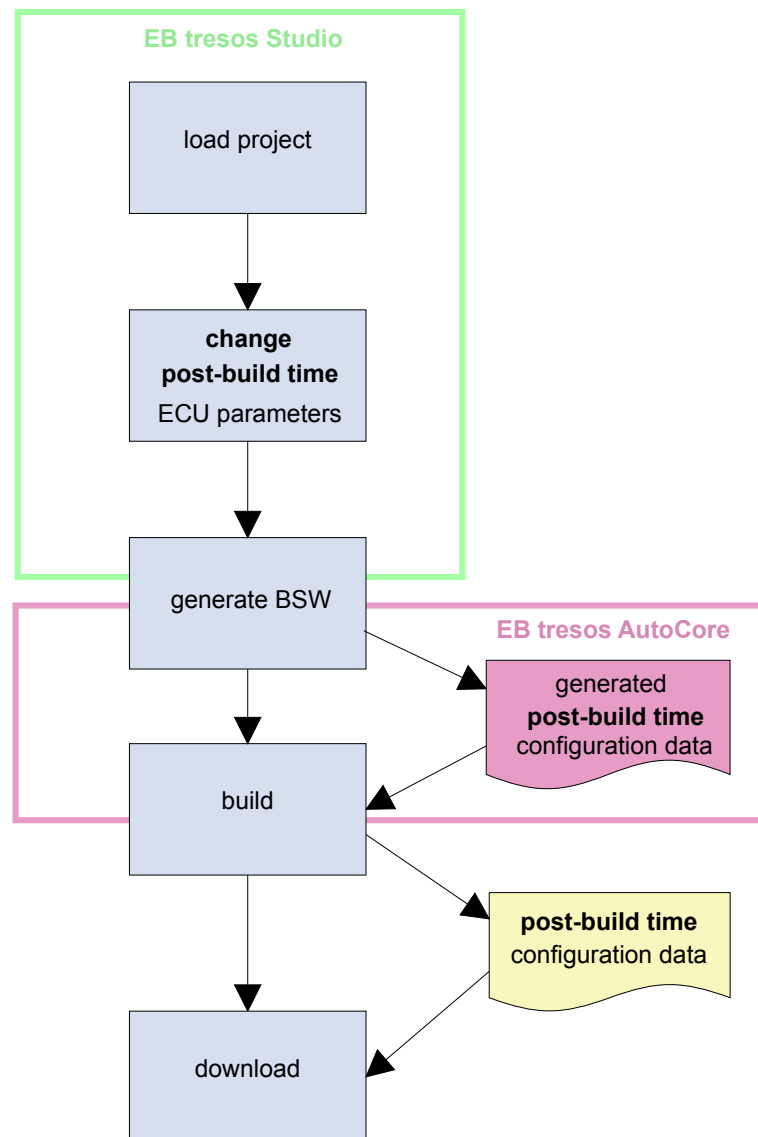


Figure 6.33. The post-build loadable use-case step 2

6.5.2.2.2. Post-build selectable concept

Post-build selectable is implemented in several EB tresos AutoCore modules. See [Section 5.2.5.2, “Post-build selectable support”](#) for a list of modules that support post-build selectable.

For these modules, it is possible to:

- ▶ define variants in the EcuC configuration
- ▶ for each variant, set different values for each configuration parameter

- ▶ generate separate configurations for each variant
- ▶ load one of these variants at startup

6.5.2.2.3. Side allocation support

As an alternative to the post-build selectable concept, EB tresos AutoCore provides side allocation support for selected configuration parameters in some modules. ECUs with side allocation use a single source concept for two ECUs, e.g. a left and a right variant. Both variants share almost all configuration parameters. Only a few configuration parameters are different. Both ECUs are flashed with the same software but the software differs during runtime depending on whether it should behave like the left or the right variant. This is determined by EB-specific callouts that must be configured in the modules that support the side allocation concept. These callouts are used to determine the variant of the software that is to run on the ECU (e.g. determined by a special value stored in EEPROM or the level of a pin on the ECU).

Refer to the module release notes in the product-specific documentation, section `EB-specific enhancements`, for information about side allocation support in the following modules:

- ▶ `CanIf`: entry with the title *CAN ID translation callout support*
- ▶ `CanTp`: entry with the title *The parameter `CanTpGeneral/CanTpDynamicNSaEnabled` was added to configure the handling of `N_SA` values*
- ▶ `Dem`: entry with the title *Support for Side Allocation*

6.5.2.2.4. Relocatable configuration data in the post-build concept

Relocatable configuration data is a specific enhancement that is strongly linked to the binary representation of the post-build configuration. The binary representation is relocatable if it can be downloaded to an arbitrary memory location. This means that the post-build configuration must not use pointers and absolute addresses to access configuration elements. Instead, the post-build configuration must use relative offsets from the configuration start address. A binary post-build configuration that is not relocatable has to be generated for the specific memory address to which it is downloaded later on.

The option to make your configuration data relocatable can be configured in any module that supports post-build configuration. See also [Section 6.5.2.2.7, “Post-build support in individual modules”](#).

If a module uses the post-build configuration manager module (`PbcfgM`) for post-build support, you can choose to make your data relocatable as part of the `PbcfgM` module configuration. See also [Section 6.5.2.2.6, “The post-build configuration manager module `PbcfgM`”](#).

6.5.2.2.5. Binary code generation in the post-build concept

EB tresos AutoCore supports direct generation of a binary representation of the post-build configuration, which can be downloaded directly to the ECU. In this case, you do not need to use a compiler or linker to create a

binary file from the post-build configuration. By supplying typical properties for the target platform (for example endianness, alignment), this generation process can be adapted automatically for all supported ECUs.

Binary generation is only available to post-build modules that use the post-build configuration manager module (`PbcfgM`) for post-build support. You can choose the binary generation option as part of the `PbcfgM` module configuration. See also [Section 6.5.2.2.6, “The post-build configuration manager module `PbcfgM`”](#).

[Figure 6.34, “The post-build loadable use-case step 2, binary generation”](#) depicts the second step in the post-build use-case adapted to show the binary generation option.

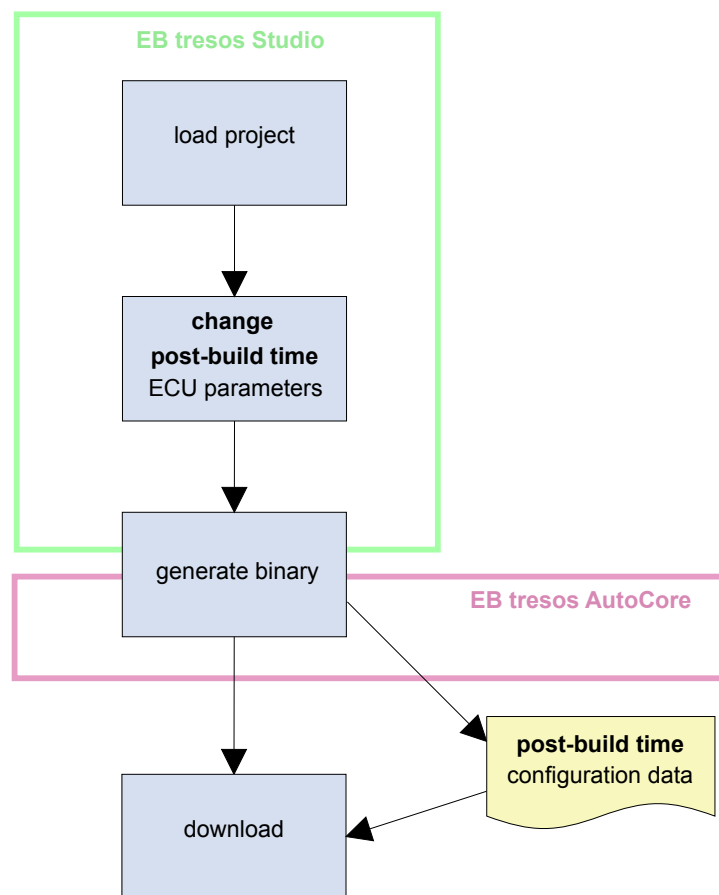


Figure 6.34. The post-build loadable use-case step 2, binary generation

6.5.2.2.6. The post-build configuration manager module `PbcfgM`

The post-build concept in the EB tresos AutoCore is managed by an EB-specific module, the `PbcfgM`. The `PbcfgM` module is supplied as part of the ACG8 Base product. You find details of this module (release notes, module references) in the ACG8 Base product-specific documentation.

A module's post-build configuration is managed by the `PbcfgM` under the following conditions:

- ▶ The module can use the `PbcfgM`, i.e. the parameter `PbcfgMSupport` is included in the module references.

and

- ▶ You add the module to the reference list in the `PbcfgM`, i.e. to the reference list parameter `PbcfgMBSWModuleRef`.

The `PbcfgM` module provides the following functionality:

- ▶ Support for the generation of a concatenated post-build configuration that contains the post-build configurations of all modules that provide post-build support and are managed by the `PbcfgM`. You can then provide this concatenated configuration to the `PbcfgM` during initialization. See also [Section 6.5.3.3, “Integration notes”](#).
- ▶ The API `PbcfgM_GetConfig()`, to supply a pointer to the post-build configuration for any module that supports the post-build concept and is managed by the `PbcfgM`. If you call the initialization function for such a module with a null pointer, the initialization function requests the configuration from the `PbcfgM` using this API. See also [Section 6.5.3.3, “Integration notes”](#).
- ▶ The API `PbcfgM_IsValidConfig()`, to check the integrity of a post-build configuration for all modules that support the post-build concept and are managed by the `PbcfgM`. The `PbcfgM` module verifies that a post-build configuration is generated with the same set of precompile time and link time parameters. The `PbcfgM` also checks if the post-build configuration is generated with the same endianness and alignment requirements and if it has the same module version number as the static software code to which it is downloaded.

This API is not used internally by the BSW modules that support the post-build concept although internal validation checks are executed during initialization.

The `PbcfgM_IsValidConfig()` API is intended to provide the integrator with the means to perform a validation check on post-build configuration data. For example, you can call this API as part of the bootloader code to verify the post-build configuration after it was downloaded to the ECU. You can also call this API during ECU start-up, e.g. before the initialization of the `PbcfgM` itself.

- ▶ Support for generation of either fixed or relocatable post-build configuration data for all modules that support the post-build concept and are managed by the `PbcfgM`. Enable or disable the `PbcfgM` parameter `PbcfgMRelocatableCfgEnable` to choose the type of configuration data generation. See also the option to manage relocatable data in individual modules in [Section 6.5.2.2.7, “Post-build support in individual modules”](#).

For modules that manage data relocation using the `PbcfgM`, the module's local option is not editable and is ignored.



- Support for the generation of either C code or binary code in Motorola S19 file format for all modules that support the post-build concept and are managed by the `PbcfgM`. Enable or disable the `PbcfgM` parameter `PbcfgMBinarySupportEnable` to choose the type of code generation.

This feature is license-protected.

NOTE



Regeneration of the `PbcfgM` required

If you regenerate selected modules only and these modules are managed by the `PbcfgM` module, you must also regenerate the `PbcfgM` module. If you do not regenerate the `PbcfgM` in this case, the respective modules fail to initialize on start-up.

6.5.2.2.7. Post-build support in individual modules

Post-build support is currently implemented in a number of ACG8 modules. You find details about the post-build support in the module references chapter `Configuration parameters` in the EB tresos AutoCore product-specific documentation.

Refer to the module references, parameter `IMPLEMENTATION_CONFIG_VARIANT`, to see if a module provides the variant `VariantPostBuild`.

Parameters included		
Parameter name	Multiplicity	
IMPLEMENTATION_CONFIG_VARIANT	1..1	

Parameter Name	IMPLEMENTATION_CONFIG_VARIANT	
Label	Config Variant	
Multiplicity	1..1	
Type	ENUMERATION	
Default value	VariantPostBuild	
Range	VariantPreCompile	
	VariantPostBuild	
Configuration class	PreCompile:	VariantPreCompile
	PreCompile:	VariantPostBuild

Figure 6.35. Supported variants of the `BswM` module

Refer to the module references, parameter `PbcfgMSupport`, to see if a module that supports the post-build concept can also use the EB post-build configuration manager module `PbcfgM`.

Parameters included	
Parameter name	Multiplicity
PbcfgMSupport	1..1

Parameter Name	PbcfgMSupport
Label	PbcfgM support
Description	Specifies whether or not the Fee can use the PbcfgM module for post-build support.
Multiplicity	1..1
Type	BOOLEAN
Default value	false
Configuration class	PublishedInformation:
Origin	Elektrobit Automotive GmbH

Figure 6.36. Post-build configuration manager support in the Fee module

6.5.2.2.7.1. Understanding the difference between the terms *implementation config variant* and *configuration class*

AUTOSAR uses two distinct concepts to define the time in the software development process that you can configure a module parameter; the *implementation config variant* and the *configuration class*.

The *implementation config variant* is an attribute of a module that determines the variants that the module may support. Possible values are `VariantPostBuild` for post-build modules, `VariantLink` for link time modules and `VariantPreCompile` for precompile time modules.

The *configuration class* is an attribute of a configuration parameter that defines at which time during the software creation process this parameter can be changed. Possible values are `PreCompile` for parameters that can be edited at precompile time, `Link` for parameters that can be edited at link time and `PostBuild` for parameters that can be edited at post-build.

TIP



The actual configuration class for a parameter depends on both the *implementation config variant* and the *configuration class*

AUTOSAR defines the possible *implementation config variants* that a module can support in the module software specification (SWS). For each of these variants, the SWS also defines the *configuration class* that shall be supported for each configuration parameter.

You can find out the actual configuration class for a parameter by checking which variant the module implements and then checking the definition of the class for that variant in the parameter attributes.

The EB tresos AutoCore modules typically provide only one *implementation config variant* per module. Therefore, the modules that support post-build configuration provide the *implementation config variant* `VariantPostBuild`.

6.5.2.2.7.2. Configuration classes of parameters in individual modules

The configuration parameters of the modules that support the post-build concept usually implement the configuration classes as specified by AUTOSAR for the implementation config variant `VariantPostBuild`. However, for technical reasons, some parameters might have different configuration classes than those specified. The configuration classes can be ordered in terms of the time in the life-cycle that they may be edited as follows (early to late): `PreCompile` -> `Link` -> `PostBuild`. If a parameter is changed to a configuration class that can be edited at a later time, it has no impact on the configuration process. This is because a parameter can always be configured at a time that is earlier than the time specified by its configuration class. If a parameter is changed to a configuration class that must be edited earlier in time, then the configuration may be more restricted. For example, a parameter changed from class `PostBuild` to class `PreCompile` can no longer be changed at post-build.

To see if a parameter can be edited at post-build, check the `Configuration class` attribute for that parameter in the module references.

See [Figure 6.37, “Description of configuration class in the module reference”](#) for an example from the ACG8 CAN Stack documentation.

[Figure 6.37, “Description of configuration class in the module reference”](#) shows the configuration parameter `CanIfTxPduCanId` which has a configuration class of `PostBuild` in the implementation config variant `VariantPostBuild` of the `CanIf` module.

Parameter Name	CanIfTxPduCanId
Description	CAN identifier of Tx L-PDUs used by the CAN Driver for CAN L-PDU transmission. Range: <ul style="list-style-type: none"> ▶ 11 bit for standard CAN identifier. ▶ 29 bit for extended CAN identifier.
Multiplicity	1..1
Type	INTEGER
Default value	0
Configuration class	PostBuild: VariantPostBuild
Origin	AUTOSAR_ECUC

Figure 6.37. Description of configuration class in the module reference

6.5.2.2.7.3. Data relocation in individual modules

Each module that supports the post-build concept and which has internal references in the configuration has an additional EB-specific parameter to enable or disable the generation of relocatable configuration data. See also [Section 6.5.2.2.4, “Relocatable configuration data in the post-build concept”](#) for a general description of this concept.

For modules that support the post-build concept together with the `PbcfgM` module, the `PbcfgM` module is responsible for managing the data relocation for all modules that are added to the `PbcfgM` reference list `PbcfgMBSwModules`. In these cases, the local data relocation configuration option is deactivated in the configuration of the individual module. See also [Section 6.5.2.2.6, “The post-build configuration manager module `PbcfgM`”](#).

6.5.2.3. Post-build support in EB tresos Studio

This section describes the support for the post-build concept provided by EB tresos Studio. Additional information about post-build configuration can also be found in the EB tresos Studio user guide and in the EB tresos Studio developer's guide.

- ▶ [Section 6.5.2.3.1, “Related requirements supported in EB tresos Studio”](#) provides a list of requirements and checks and how they are supported.

6.5.2.3.1. Related requirements supported in EB tresos Studio

In addition to the requirements of AUTOSAR 4.0.3, EB tresos Studio supports enforcement of the following requirements which are based on AUTOSAR 4.1.1 and selected RfCs. The requirements are enforced either as a VSMD check or as an online check during configuration:

- ▶ **TPS_ECUC_08000:** The attribute `postBuildChangeable` specifies if the number of instances of this `EcucContainerDef` may be changed in post-build selectable and post-build loadable time in the `ECU Configuration Value Description`.

Online check: For post-build selectable configuration, setting the attribute `postBuildChangeable` to true means that different number of instances of this container may exist in different configuration sets. For post-build loadable configuration, setting the attribute `postBuildChangeable` to true means that new instances of this container may be introduced or existing instances removed in the new configuration set.

- ▶ **constr_05500:** The attribute `postBuildChangeable` is applicable only to `EcucContainerDefs` which have `upperMultiplicity` greater than `lowerMultiplicity` and `upperMultiplicity` is greater than 1.

VSMD check (new check): This constraint on the applicability of the `postBuildChangeable` attribute is checked by EB tresos Studio. See also the EB tresos Studio developer's guide chapter `Additional VSMD rules regarding post-build enabled configuration modules` and the EB tresos Studio user guide chapter `Checking vendor-specific against standard module definitions`.

- ▶ **constr_05504:** Only instances of `EcucContainerDefs` with the attribute `postBuildChangeable` set to true which are exclusively referenced by `EcucAbstractReferenceDefs` with `PostBuild` configuration class can be removed at post-build time.

Online check: After removal of a container at post-build time, EB tresos Studio checks during configuration or during generation that no precompile time or link time parameter references the container.

- ▶ **TPS_ECUC_08003:** An `EcucContainerDef` may have the attribute `postBuildChangeable` set to true, even if one or more of its aggregated `EcucContainerDefs` in the role `subContainer` have the attribute `postBuildChangeable` set to false.

VSMD check (relaxation of existing check): This constraint on the applicability of the `postBuildChangeable` attribute is checked by EB tresos Studio. See also the EB tresos Studio developer's guide chapter `Additional VSMD rules regarding post-build enabled configuration modules` and the EB tresos Studio user's guide chapter `Checking vendor-specific against standard module definitions`.

Online check: If a container A has the attribute `postBuildChangeable` set to true and in its subcontainer B it is set to false, it is not possible to add a new instance b2 of subcontainer B to the existing container instance a1 of A in post-build. However, it is possible to add a new instance a2 of the `postBuildChangeable` container A together with a new instance b2 of its subcontainer B.

- ▶ **TPS_ECUC_08001:** `EcucParameterDefs` and `EcucAbstractReferenceDefs` within an `EcucParamConfContainerDef` and its aggregated `EcucParamConfContainerDefs` in the role `subContainer` which have the attribute `postBuildChangeable` set to true may have either `PreCompile`, `Link` or `PostBuild` configuration class.

Online check (relaxation of existing check): It is now allowed to add subcontainers with parameters that have configuration classes other than `PostBuild` to containers that have the attribute `postBuildChangeable` set to true.

- ▶ **TPS_ECUC_08002:** If a new `EcucParamConfContainerDef` instance is introduced according to the **TPS_ECUC_08000** in a post-build loadable configuration set, each `EcucParameterValue` and `EcucReferenceValue` within that `EcucParamConfContainerDef` instance may be assigned a new value regardless of its configuration class (`PreCompile`, `Link` or `PostBuild`). Also its aggregated `EcucParamConfContainerDefs` instance and its aggregated `EcucParamConfContainerDefs` instanced in the role `subContainer` may be assigned a new value regardless of its configuration class (`PreCompile`, `Link` or `PostBuild`).

Online check (relaxation of existing check): Parameters that have configuration classes other than `PostBuild` may now be assigned new values within subcontainers that are added at post-build.

- ▶ **TPS_ECUC_08004:** If there exist multiple `EcucParameterValues` inside one `EcucContainerValue` (i.e. the defining `EcucDefinitionElement.upperMultiplicity` is greater than `EcucDefinitionElement.lowerMultiplicity`), both value and multiplicity of this `EcucParameterValue` may be changed at post-build. Also the configuration class of the defining `EcucParameterDef` must be `PostBuild`.

Online check: This means that it is possible to change the value of a parameter. It is also possible to introduce new instances of this parameter or remove some instances in different post-build loadable and selectable configuration sets.

- ▶ **TPS_ECUC_08005:** In case `supportedConfigVariant` of a `EcucModuleDef` equals `VariantPostBuild` and the attribute `postBuildChangeable` of an `EcucContainerDef` is not defined in the `StMD`, it shall be defined in the `VSMD` for all `EcucContainerDefs`. The `EcucContainerDefs` must have `upperMultiplicity` greater than `lowerMultiplicity` and `upperMultiplicity` must be greater than 1. This includes vendor specific `EcucContainerDefs`.

VSMD check (new check)

- ▶ **TPS_ECUC_08006:** In case `supportedConfigVariant` of a `EcucModuleDef` equals `VariantPostBuild` and an `EcucContainerDef` in the `StMD` has the attribute `postBuildChangeable` set to false, the corresponding `VSMD` may change it to true if its `upperMultiplicity` is greater than `lowerMultiplicity` and `upperMultiplicity` is greater than 1.

VSMD check (new check)

- ▶ **TPS_ECUC_08007:** In case `supportedConfigVariant` of a `EcucModuleDef` equals `VariantPostBuild` and an `EcucContainerDef` in the StMD has the attribute `postBuildChangeable` set to true, it shall be set to true in the corresponding VSMD as well.

VSMD check (new check)

- ▶ **TPS_ECUC_06051:** The `implementationConfigClass` of an `EcucParameterDef` or `EcucAbstractReferenceDef` in VSMD shall be the same or higher (where `PreCompile` configuration class is considered to be the lowest and `PostBuild` the highest) than in StMD with respect to the selected subset defined by the actually implemented `supportedConfigVariant` if the scope of the `EcucParameterDef` or `EcucAbstractReferenceDef` in StMD is ECU.

VSMD check (relaxation of existing check): A configuration parameter may implement a higher configuration class than specified.

NOTE



`constr_05501` and `constr_05502` are not enforced

`constr_05501` and `constr_05502` are not enforced by EB tresos Studio but at runtime by the signature check in the `<module>_IsValidConfig()` function.

NOTE



`constr_05503` is not implemented

`constr_05503` is not implemented. You must ensure the consistency of symbolic name value macros.

6.5.3. Using post-build support in EB tresos AutoCore

The section describes how you can start using the post-build support in your EB tresos AutoCore project.

- ▶ [Section 6.5.3.1, “The post-build configuration workflows”](#) describes the EB tresos AutoCore workflows that show you how to set up and use post-build support.
- ▶ [Section 6.5.3.2, “The workflow for variant handling”](#) describes how variants are configured and managed.
- ▶ [Section 6.5.3.3, “Integration notes”](#) provides you with further information that is important for integrating the modules that provide post-build support in your project.

6.5.3.1. The post-build configuration workflows

This section provides an overview of the three workflows that are related to the post-build configuration. These workflows are provided in the EB tresos AutoCore Generic 8 Base product. To use these workflows, see the EB tresos AutoCore Generic 8 Base product documentation.

- ▶ Workflow post-build setup: add post-build support to an existing project.

The post-build setup workflow covers the following steps:

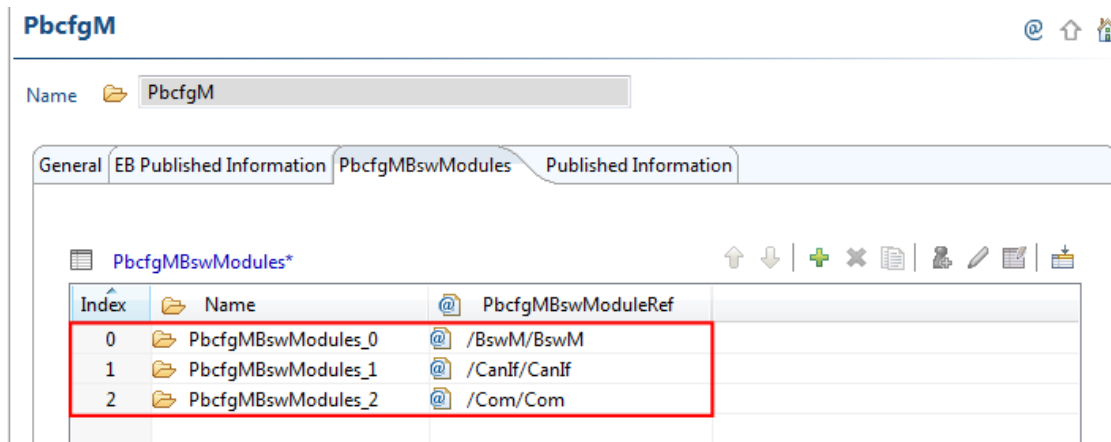


Figure 6.38. Modules in `PbcfgM` reference list

- ▶ Add the `PbcfgM` module to an existing project.
- ▶ Configure the `PbcfgM` module to reference all modules for which it shall provide support. For example, the access to a module's post-build configuration.
- ▶ Configure settings for relocation of configuration data, configuration start address and binary code generation.
- ▶ Workflow post-build only: modify an existing project to solely build the post-build configuration.

The post-build only workflow covers the following steps:

- ▶ Disable the code generation for all modules that do not support post-build configuration.
- ▶ Store the generated pre-build time configuration files of the disabled modules for later use.
- ▶ Remove all remaining generated configuration files.
- ▶ Create a binary from the post-build configuration either with the compiler tool chain or directly via EB tresos Studio when binary code generation is enabled.
- ▶ Workflow post-build update: update the post-build configuration.

The post-build update workflow covers the following steps:

- ▶ Update post-build parameters in your configuration project.
- ▶ Generate a post-build configuration again.

6.5.3.2. The workflow for variant handling

For a detailed description on how to work with variants in your project, refer to the *EB tresos Studio user guide*, chapter 6.13.4 "Recommended workflow".

6.5.3.3. Integration notes

While precompile time and link time configuration parameters are part of the fixed software code or are linked to it, the post-build configuration has to be passed to the initialization function of the respective modules at run-time. The initialization sequence to use depends on whether or not a module that supports post-build also uses the `PbcfgM` module for support.

Refer to the module references, parameter `PbcfgMSupport`, to see if a module that supports the post-build concept can also use the EB post-build configuration manager module `PbcfgM`.

6.5.3.3.1. Initializing modules without `PbcfgM` support

If you use a module that supports the post-build concept but which does not use the `PbcfgM` module, you must take care that a valid configuration is provided during initialization.

You call the module's initialization function with a pointer to your valid post-build configuration. For example:

```
Modl_Init(USERS_VALID_CONFIG_PTR);
```

It is the user's responsibility to ensure that a valid post-build configuration is provided (e.g. endianness, alignment). Further internal checking does not take place within the module's initialization function.

Note that you can not use the `PbcfgM` API function `PbcfgM_IsValidConfig()` to run a validation check in this case.

6.5.3.3.2. Initializing modules with `PbcfgM` support

When you use a module that supports the post-build concept together with the `PbcfgM` module, you can choose to initialize the module in one of the following ways:

- ▶ Use the `PbcfgM` support to provide a valid configuration to the initialization function of the module. To do this, call the module's initialization function with a null pointer:

```
Modl_Init(NULL_PTR);
```

- ▶ Provide a pointer to a valid post-build configuration to the initialization function of the module. To do this, call the module's initialization function with a valid pointer that is not a null pointer:

```
Modl_Init(USERS_VALID_CONFIG_PTR);
```

The module's initialization function performs a validity check (e.g. endianness, alignment) on the post-build configuration in both of these cases.

6.5.3.3.2.1. Initializing the `PbcfgM`

If you choose to use the `PbcfgM` module, it must be initialized before all other modules that use it. To achieve this, you can add the `PbcfgM` to the `Module Initialization List One` (`EcuMDriverInitListOne`) as described in the EB tresos AutoCore Generic Mode Management documentation. Each post-build module that is referenced by the `PbcfgM` module can then be initialized by calling its initialization function and passing the null pointer to it. See for example the following code:

```
PbcfgM_Init( &PBCFGM_CONFIG_NAME );  
Mod1_Init(NULL_PTR);  
Mod2_Init(NULL_PTR);  
Mod3_Init(NULL_PTR);
```

6.5.3.3.3. Configuration of post-build RAM

Post-build RAM refers to the volatile memory that is used to store post-build-dependent run-time data. The amount of post-build RAM required depends on the post-build configuration. If an update or switch of the post-build configuration is planned, it must be ensured that for all expected configurations sufficient post-build RAM is available. Most modules have a simple relationship between the local configuration and the amount of run-time memory that is required, usually this dependency is modelled by just one or two pre-compile time parameters (e.g. maximum number of channels). Then it suffices to configure this parameter accordingly (e.g. initially configure a larger maximum number of channels in case an additional channel shall be added at post-build time). But some modules (currently `PduR`, `Com`, `FrIf` and `SoAd`) have a complex dependency which cannot easily be modelled. For these modules the available post-build RAM can be configured at link time in bytes or the module's generator can calculate it based on the current configuration.

If one of the modules `PduR`, `Com`, `FrIf` or `SoAd` are used in a project and if a post-build update of the configuration of these modules is planned then it must be ensured that sufficient post-build RAM is configured. This can be done by following these guidelines:

- ▶ Make sure that these parameters are enabled and configured: `ComDataMemSize`, `PduRMemorySize`, `FrIfDataMemSize` and `SoAdDataMemSize`
- ▶ Make sure that all configurations - the initial configuration and all post-build configurations - can be generated with the configured post-build RAM size. A generation error is reported if the configured post-build RAM is too small.
- ▶ The exact size of required post-build RAM can be found by checking the size of the generated link time variables: `Com_DataMem`, `PduR_Mem`, `FrIf_Mem` and `SoAd_Mem`

NOTE



Parameters for configuration of post-build RAM have configuration class *link* and therefore cannot be changed at post-build time

You must configure sufficient memory at pre-compile time or link time. Attempting to change the parameter at post-build time will result in invalid code which the `PbcfgM` module will reject during the update process.

6.6. AUTOSAR 4.x support

6.6.1. Overview

EB tresos AutoCore Generic 8 supports project development that includes modules from AUTOSAR 4.0 and selected interfaces of AUTOSAR 4.2 and 4.3. This chapter provides the following information:

- ▶ [Section 6.6.2, “Background information”](#) explains the concepts of the AUTOSAR 4.x support within EB tresos AutoCore Generic 8.
- ▶ [Section 6.6.3, “Configuring the required service APIs”](#) shows how to configure a BSW service module to provide an AUTOSAR 4.x-compliant API.
- ▶ [Section 6.6.4, “Generating AUTOSAR 3.2/4.0/4.x schema-compliant BSW SWC descriptions”](#) provides instructions on how to generate AUTOSAR 4.x-compliant SWC descriptions of BSW service modules.

NOTE



AUTOSAR 3.2 support

Some modules also offer AUTOSAR 3.2 support in EB tresos AutoCore Generic 8. For these modules, the module references provide API information for both AUTOSAR 3.2 and for AUTOSAR 4.x. Only API functions that are used via the `Rte` are affected. You find the module references in the product-specific documentation provided with the EB tresos AutoCore.

To understand the details of the following sections, you need to be familiar with basic concepts of the `Rte`. See the EB tresos AutoCore Generic RTE documentation for information about how to configure the `Rte`.

6.6.2. Background information

This chapter describes the basic concepts of the AUTOSAR 4.x support as it is implemented in EB tresos AutoCore Generic 8.

- ▶ [Section 6.6.2.1, “EB tresos AutoCore Generic 8 in an AUTOSAR 4.x environment”](#) gives an overview of how EB tresos AutoCore Generic 8 is embedded in an AUTOSAR 4.x context.

- ▶ [Section 6.6.2.2, “Overview of the target code”](#) explains what AUTOSAR 4.x support means with regard to the target code.
- ▶ [Section 6.6.2.3, “Interface options for BSW service modules”](#) explains the interface concept used by BSW service modules to provide AUTOSAR 4.x support.

If you are already familiar with the basic concepts of the AUTOSAR 4.x support within EB tresos AutoCore Generic 8, you may want to skip this chapter and proceed to the instruction chapters starting with [Section 6.6.3, “Configuring the required service APIs”](#).

6.6.2.1. EB tresos AutoCore Generic 8 in an AUTOSAR 4.x environment

[Figure 6.39, “Conceptual overview of AUTOSAR 4.x support”](#) shows how the EB tresos AutoCore Generic 8 modules are integrated with an AUTOSAR 4.2 or 4.3 project environment.

The EB tresos AutoCore Generic 8 stack, which consists of the `Rte` and the AUTOSAR BSW modules, is implemented according to AUTOSAR 4.0. To use AUTOSAR 4.2/4.3 software components, import the corresponding system description and the SWC descriptions as described in the EB tresos AutoCore Generic RTE documentation. During this step, EB tresos Studio converts these descriptions to an internal representation that is compatible with AUTOSAR 4.0. When you use EB tresos Studio, you can configure EB tresos AutoCore Generic 8 to generate an AUTOSAR 4.x-compliant API for the BSW service modules. EB tresos Studio creates AUTOSAR 4.x wrapper functions that can be used by your client application SWCs.

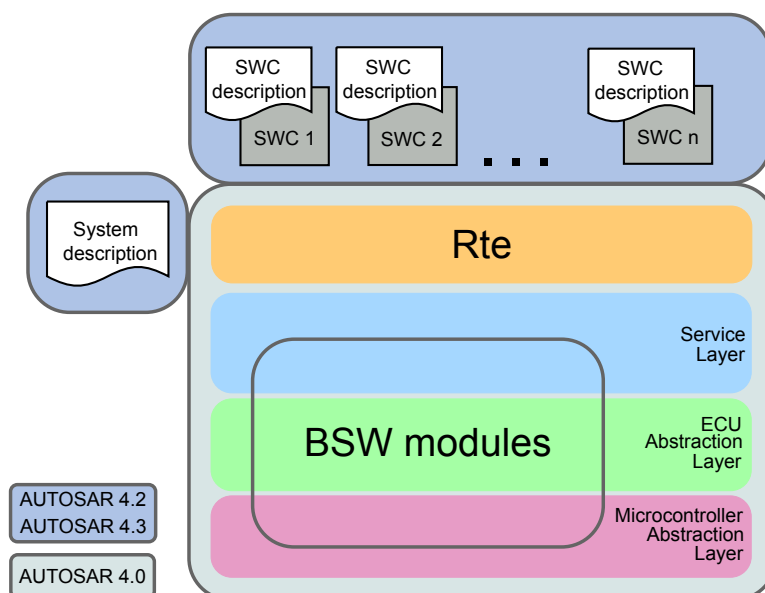


Figure 6.39. Conceptual overview of AUTOSAR 4.x support

6.6.2.2. Overview of the target code

[Figure 6.40, “Schematic overview of the AUTOSAR 4.x target code”](#) depicts how the AUTOSAR 4.x API is supported by the EB tresos AutoCore Generic 8 components. The `Rte` and the BSW service modules provide wrapper functions which map the AUTOSAR 4.x API to the corresponding AUTOSAR 4.0 API implementation. These wrapper functions can be used by your AUTOSAR 4.x client application SWCs. See the EB tresos AutoCore Generic RTE documentation for instructions and information about using the AUTOSAR 4.x wrapper provided by the `Rte`.

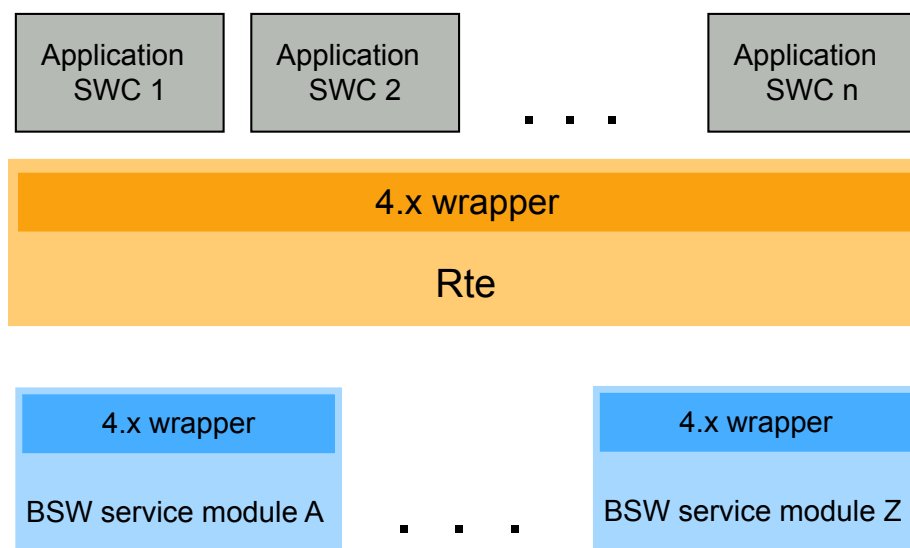


Figure 6.40. Schematic overview of the AUTOSAR 4.x target code

6.6.2.3. Interface options for BSW service modules

Client application SWCs can choose whether to use AUTOSAR 4.0, 4.2, 4.3, or the configurable default interface of a BSW service module. Depending on your project requirements, you can use a combination of these options or simply use the default AUTOSAR 4.0 for all interfaces. For information on how to configure the required interface to be generated, see [Section 6.6.3, “Configuring the required service APIs”](#). As depicted in [Figure 6.41, “Schematic overview of interfaces for different AUTOSAR versions”](#), the AUTOSAR version-specific interface names are infixed with a version specifier (e.g. `ASR42`) to make them distinguishable.

In addition, a non-infixed default interface exist. You can configure this default interface to comply with AUTOSAR 4.0, 4.2, 4.3, or to not be available at all. You must adapt the client application SWCs to use the correct interface, depending on the configuration you have chosen.

NOTE



Multiple interfaces

You can configure the BSW service modules to provide AUTOSAR 4.0, 4.2, and 4.3 interfaces simultaneously. This applies to selected modules.

However, require ports for a BSW service module are only available for the selected default AUTOSAR version.

TIP



Configure the default service API

If you use the configurable default interface, you do not need to change the client application SWCs.

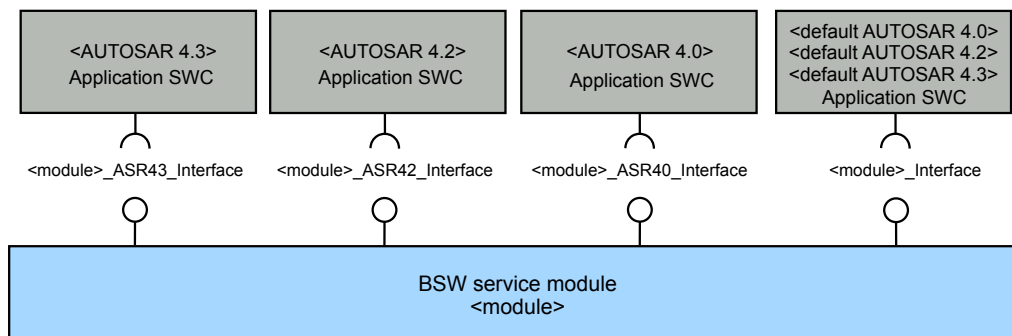


Figure 6.41. Schematic overview of interfaces for different AUTOSAR versions

6.6.2.4. EB tresos AutoCore Generic 8 modules that provide 4.x support

The following modules provide multiple versions of selected AUTOSAR 4.x interfaces in EB tresos AutoCore Generic 8:

Module	EB product	Supported AUTOSAR version		
		4.0	4.2	4.3
Dcm	ACG8 Diagnostic Stack	x	x	x
Dem	ACG8 Diagnostic Stack	x	x	-
Det	ACG8 Base	x	x	-
Dlt	ACG8 DLT	x	x	-
NvM	ACG8 Memory Stack	x	x	-

In each of these modules, the module references provide API information for AUTOSAR 4.0 and for the supported AUTOSAR 4.x version. Only API functions that are used via the `Rte` are affected. You find the module references in the product-specific documentation provided with the EB tresos AutoCore.

[Figure 6.41, “Schematic overview of interfaces for different AUTOSAR versions”](#) shows a schematic overview of the interfaces and the API naming conventions. For example, in the `Dem` module, you find the functions `Dem_AS42_GetEventFreezeFrameData()` and `Dem_GetEventFreezeFrameData()` listed in the module references.

NOTE



Infix names for new API functions in AUTOSAR 4.2

New API functions that exist in AUTOSAR 4.2 only have one infix name available.

6.6.3. Configuring the required service APIs

This chapter describes how you must configure the BSW service modules in order to provide the required AUTOSAR 4.x and/or default service APIs.

NOTE



Common service API configuration

Configuring the service APIs of a BSW service module, e.g. `NvM`, is common to all BSW service modules.

6.6.3.1. Use cases

Depending on your project requirements, the following settings are recommended:

- ▶ All SWCs shall use one standard AUTOSAR interface: configure the default to be either AUTOSAR 4.0, AUTOSAR 4.2, or AUTOSAR 4.3, leave other options disabled.
- ▶ Most SWCs shall use AUTOSAR 4.0, some shall use AUTOSAR 4.2 or AUTOSAR 4.3: configure the default to be AUTOSAR 4.0 and activate the option to enable the AUTOSAR 4.2 or 4.3 interface.
- ▶ Most SWCs shall use AUTOSAR 4.2 or AUTOSAR 4.3, some shall use AUTOSAR 4.0: configure the default to be AUTOSAR 4.2 or 4.3, and activate the option to enable the AUTOSAR 4.0 interface.

6.6.3.2. Configuring steps

To configure the required service API of a BSW service module:

- ▶ Select the **General** tab in the **Editor** view of the BSW service module, e.g. in the `NvM`.

- ▶ Activate **Enable Rte usage** to enable the use of the `Rte` and additional *Service API parameters*.
- ▶ Navigate to the **Service API Parameters** section:

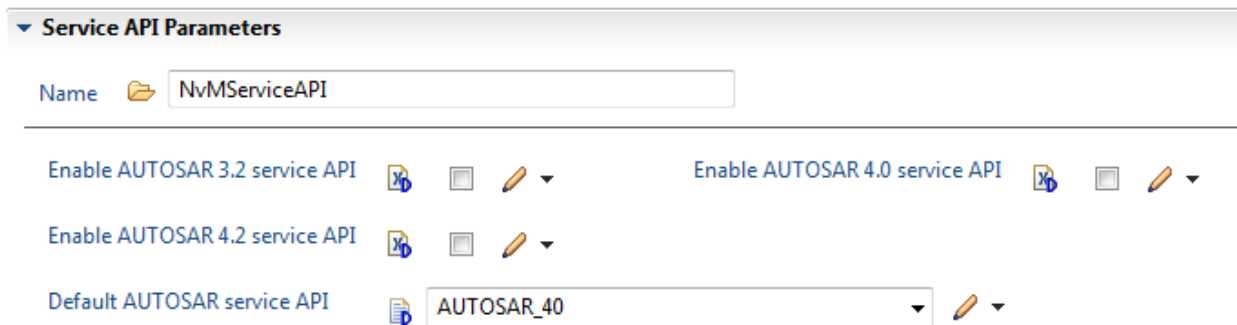


Figure 6.42. **Service API Parameters** section of the **Editor** view

- ▶ Activate **Enable AUTOSAR 4.0 service API** if you want the BSW service module to provide an AUTOSAR 4.0 interface.

Activate **Enable AUTOSAR 4.2 service API** if you want the BSW service module to provide an AUTOSAR 4.2 interface.

Activate **Enable AUTOSAR 4.3 service API** if you want the BSW service module to provide an AUTOSAR 4.3 interface.

At **Default AUTOSAR service API**:

- ▶ Select **AUTOSAR_40** to set AUTOSAR 4.0 as the default interface.
- ▶ Select **AUTOSAR_42** to set AUTOSAR 4.2 as the default interface.
- ▶ Select **AUTOSAR_43** to set AUTOSAR 4.3 as the default interface.

6.6.4. Generating AUTOSAR 3.2/4.0/4.x schema-compliant BSW SWC descriptions

In EB tresos Studio, you can generate AUTOSAR 3.2, 4.0, or 4.x compliant BSW SWC descriptions. Select the desired schema compliance to trigger the generation mode of the specific SWC description.

Under **Project/Build Project/** select one of the following menu items to trigger the generation of SWC descriptions:

- ▶ **generate_asr32_swcd** to generate AUTOSAR 3.2-compliant BSW SWC descriptions
- ▶ **generate_asr40_swcd** to generate AUTOSAR 4.0-compliant BSW SWC descriptions
- ▶ **generate_swcd** to generate AUTOSAR default BSW SWC descriptions

NOTE



Module support for AUTOSAR 4.x

All modules that have support for AUTOSAR 4.x only offer a generation mode for compliance with AUTOSAR 4.0/4.x.

Therefore, only **generate_swcd** is available.

6.7. Measurement and calibration

6.7.1. Overview

This chapter explains the concept of measurement and calibration as defined by the Association for Standardisation of Automation and Measuring Systems (ASAM) and AUTOSAR and describes how EB tresos AutoCore Generic and EB tresos Studio support measurement and calibration.

ASAM is an association of car manufacturers, suppliers and engineering service providers from the automotive industry. ASAM coordinates the development of technical standards that define protocols, data models, file formats and application programming interfaces (APIs) for use in the development and testing of automotive electronic control units. For more details about the XCP protocol and standards, refer to the ASAM specifications (www.asam.net).

The AUTOSAR standard provides mechanisms to abstract and define measurement and calibration data as part of the software component description or basic software description. For more details refer to AUTOSAR specifications: [\[8\]](#), [\[9\]](#) and [\[10\]](#).

- ▶ [Section 6.7.2, “Background information”](#) explains the concepts of measurement and calibration.
- ▶ [Section 6.7.3, “Using measurement and calibration”](#) describes how to use the measurement and calibration features in EB tresos AutoCore Generic products.

6.7.2. Background information

This section provides background information on the following topics:

- ▶ [Section 6.7.2.1, “Functional overview”](#) explains the concept of measurement and calibration in ASAM and how this is applied in AUTOSAR
- ▶ [Section 6.7.2.2, “Modules involved in measurement and calibration”](#) explains the modules that are involved in measurement and calibration and the role they play.
- ▶ [Section 6.7.2.3, “Modelling measurement and calibration data”](#) explains how measurement and calibration data must be modelled in the basic software module description and/or software component description.

6.7.2.1. Functional overview

This section provides a brief overview of measurement and calibration concepts.

6.7.2.1.1. Measurement

In order to monitor system behavior you can measure internal data or parameters of the ECU. This means that the value of the relevant data or parameters is sampled by a slave and sent on a communication bus to the master.

6.7.2.1.2. Calibration

Calibration means adjusting ECU SW characteristics and parameters in order to fulfill tasks and to optimize the behavior and functions that are performed on a system.

Calibration operations can be done manually by a calibration engineer or they can be automated by using specialized external SW tools. Such tools require a detailed description of the ECU internal variables and the characteristics that are to be accessed.

The calibration can be done offline or online. The difference between the two possibilities consists in i) doing measurement while ECU is running and calibration while ECU is offline or ii) doing measurement and calibration in parallel online. The choice of how to do calibration is project specific.

Measurement and calibration are normally only used during the development phase of ECUs and it requires address oriented READ and/or WRITE access to ECU internal variables.

6.7.2.1.3. Generation of A2L files

ASAM defines a standard to specify and describe the data in a measurement and calibration (MC) system that is independent from ECU-internal formats. The standard defines description of the data, description of the memory segments in the ECU, allocation of data to addresses, the type of memory, and data accesses. The standard also describes the interface between the MC system and the ECU for read and write access. As a result, the ASAM MCD-2 MC description contains all information in one place for access, modification, interpretation, and display of ECU-internal variables. The data description is written in a structured ASCII format known as A2L format (*.a2l). This format can easily be parsed and imported.

EB tresos AutoCore supports generation of A2L files according to ASAM MCD-2 MC.

6.7.2.2. Modules involved in measurement and calibration

This section lists the EB tresos AutoCore Generic (ACG) products that support measurement and calibration and briefly explains their main functionality. You can find detailed information in the respective product-specific user documentation.

► ACG8 RTE

The ACG8 RTE supports measurement and calibration according to the AUTOSAR standard. Measurement and calibration data can be defined in the BSW module description (BSWMD) or software component description (SWCD) as described [Section 6.7.2.3, “Modelling measurement and calibration data”](#) below. The `Rte` generates the ARXML file with MC support data. MC data is used by the ACG8 A2L generator as depicted in [Figure 6.43, “Generation of A2L file”](#).

You find detailed information about support for measurement and calibration in the ACG8 RTE product user documentation.

► ACG8 A2L generator

The ACG8 A2L generator reads measurement and calibration data, converts it to the ASAM MCD-2 MC standard format and generates an initial A2L file containing the symbol names. Address information is obtained from the map file and is generally merged with the initial A2L file to create a final A2L file by the XCP master. You can use the ACG8 A2L generator after project generation in EB tresos Studio.

[Figure 6.43, “Generation of A2L file”](#) provides a simplified overview of the data flow during the generation of the A2L file.

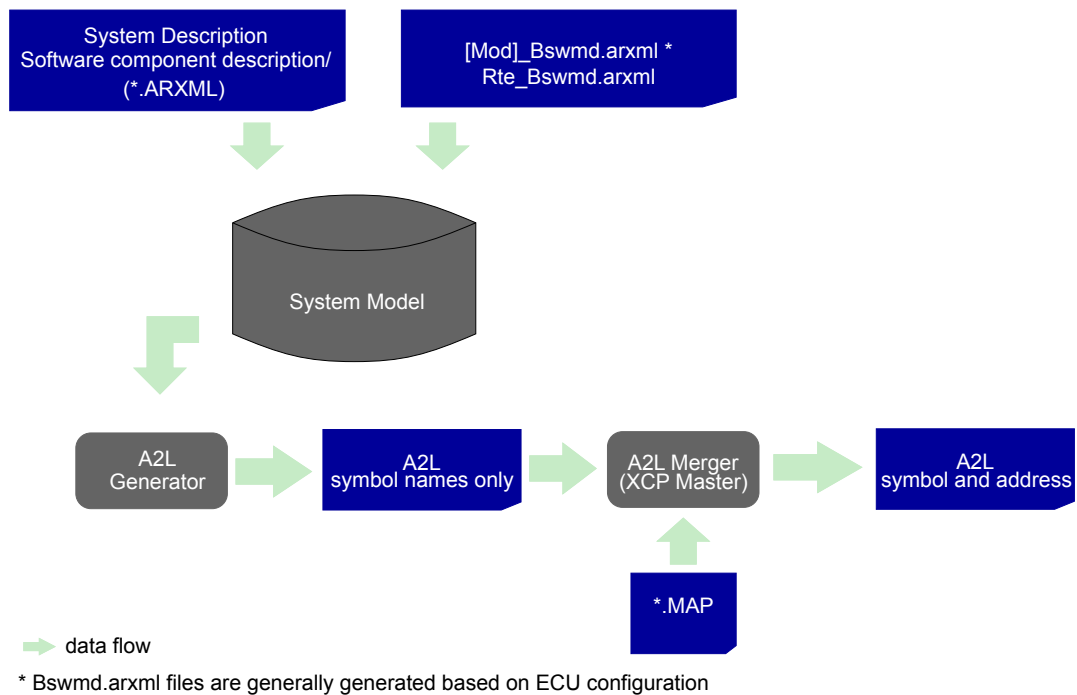


Figure 6.43. Generation of A2L file

► ACG8 XCP

ACG8 XCP is implemented according to the Universal measurement and calibration protocol defined by the Association for Standardization of Automation and Measuring Systems (ASAM). You can use ACG8 XCP on different bus types, for example FlexRay, CAN or Ethernet. ACG8 XCP supports the following basic features:

- Synchronous data acquisition and stimulation
- Online memory calibration (read / write access)
- Calibration data page initialization and switching
- Flash Programming for ECU development purposes

[Figure 6.44, “Measurement and calibration setup”](#) provides a simplified overview of a measurement and calibration setup.

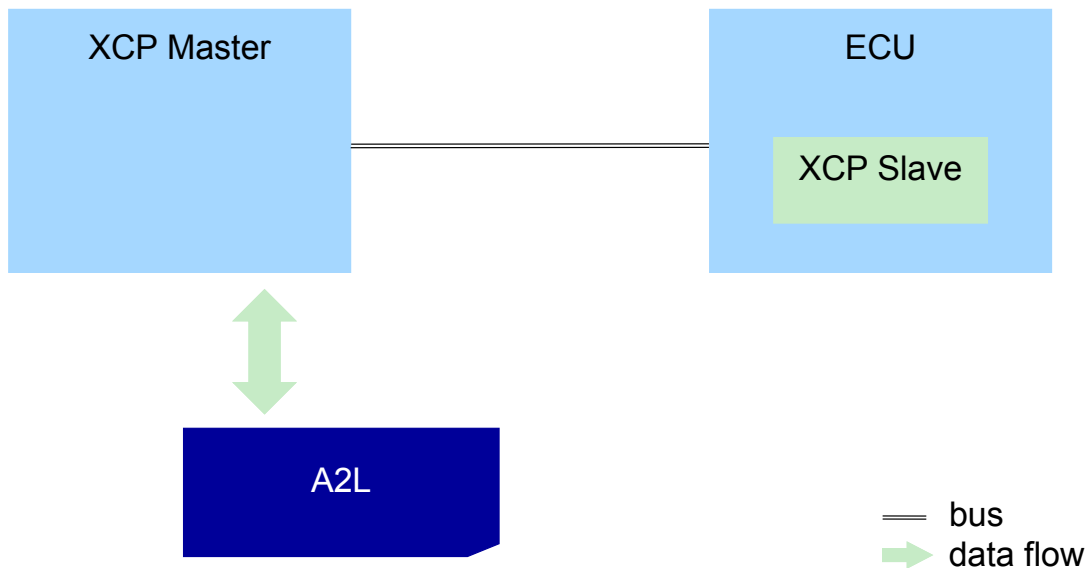


Figure 6.44. Measurement and calibration setup

6.7.2.3. Modelling measurement and calibration data

For one ECU, AUTOSAR allows you to declare measurement and calibration data (MC-data) in the basic software module description or in the software component description. However, the measurement and calibration tool e.g. XCP master, generally uses another representation e.g. an A2L file.

According to AUTOSAR methodology an additional intermediate ARXML work product should be provided, the so-called MC Support Data, which is produced rather late in the ECU configuration process e.g. after project generation using EB tresos Studio.

6.7.3. Using measurement and calibration

This section describes the steps that have to be followed in order to use measurement and calibration features in your project:

- ▶ [Section 6.7.3.1, “Configuring the Xcp module”](#) describes the most important steps to be followed for configuring and integrating the `Xcp` module in your project.
- ▶ [Section 6.7.3.2, “Configuring measurement and calibration data”](#) explains how to use and describe measurement and calibration data.

6.7.3.1. Configuring the Xcp module

This chapter describes the most important steps that have to be followed to configure the `Xcp` module. For a detailed description of all features and configuration parameters, refer to the ACG8 XCP user documentation.

When configuring the `Xcp` module you should consider project specific requirements and constraints: the number and type of bus interfaces to be used, the number of PDUs that can be used, the size of data that has to be exchanged between the master and the slave, how often the data has to be transmitted/received.

The major steps you should follow when configuring the `Xcp` module are:

- ▶ Configure the communication modules to be used for the lower layer of the `Xcp` (communication driver and bus-specific interface).
- ▶ Configure the PDUs to be used by `Xcp`. At least one transmission and one reception PDU has to be configured for each connection. Add a new configuration set and in the tab **PDU Information** configure the PDUs.
- ▶ Configure the connection to be used, assign the PDUs to the connection and configure the types of packages to be supported by each PDU. At least one connection has to be configured. In order to configure a connection in the tab **Connection Information** add a new connection and go to tabs **Tx PDUs to Connection mapping** and **Xcp Rx PDUs to Connection mapping** for assigning and configuring connection PDUs.
- ▶ Depending on project requirements, in the tab **Optional Features** you can enable additional features (also support for calibration commands) or in the tab **DAQ/STIM Setup** you can enable and configure data acquisition and data stimulation.

6.7.3.2. Configuring measurement and calibration data

All basic software modules or software components can declare measurement and calibration data in the basic software module description or in the software component description.

To enable measurement and calibration data you must add the root element `MC-SUPPORT` to the `IMPLEMENTATION` of your module or component.

To define measurement and calibration data, for each data element you must add an `MC-DATA-INSTANCE` and describe the data within this entry.

[Figure 6.45. “Modelling measurement and calibration data”](#) provides a simplified overview of modelling measurement and calibration data.

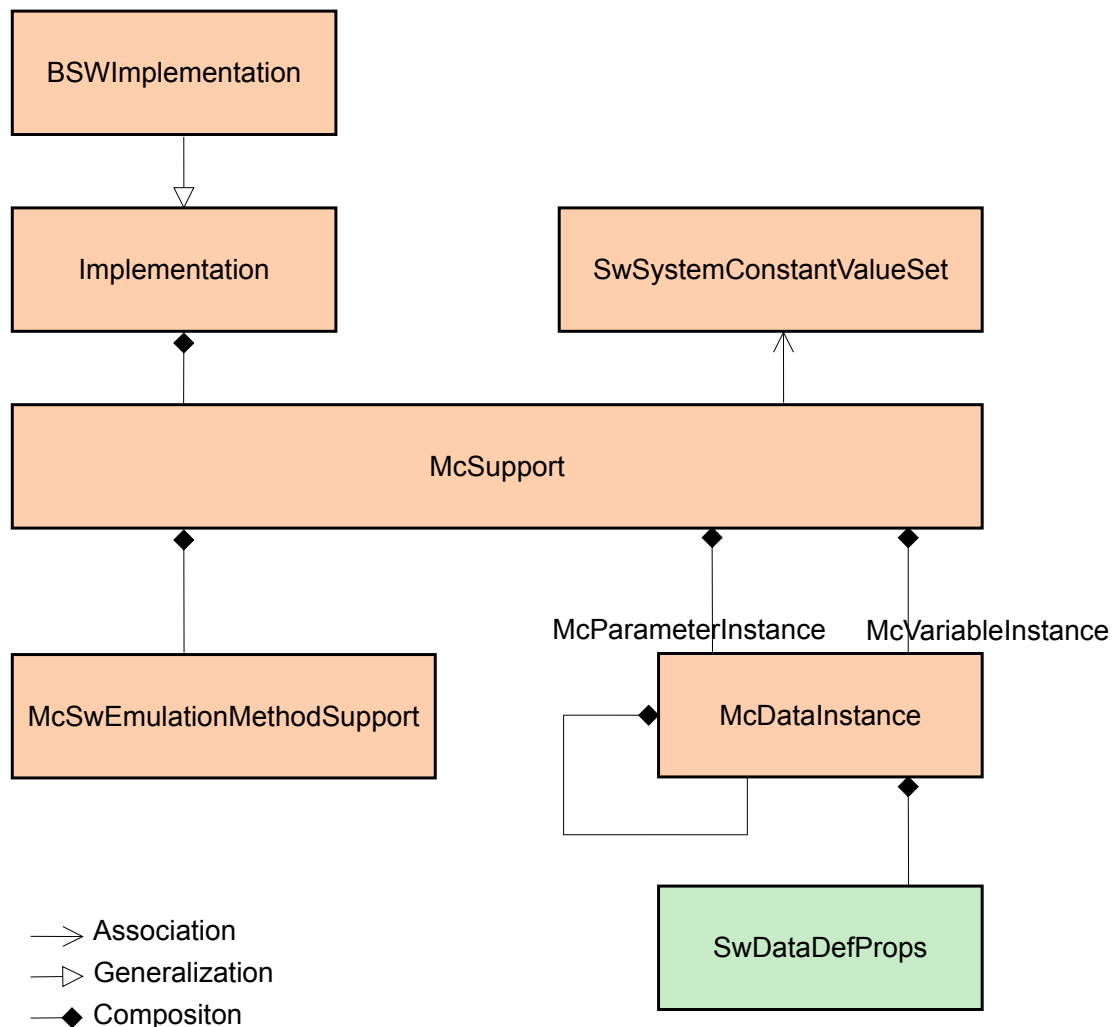


Figure 6.45. Modelling measurement and calibration data

6.8. Multi-core support

6.8.1. Overview

This chapter explains the multi-core support in EB tresos AutoCore Generic (ACG). This includes the distribution of application software components as well as the distribution of selected basic software modules on multiple cores.

- [Section 6.8.2, “Background information”](#) explains the aspects of multi-core support in AUTOSAR and EB tresos AutoCore Generic.

- ▶ [Section 6.8.3, “Application support”](#) describes the support for distribution of application software components on multiple cores.
- ▶ [Section 6.8.4, “BSW distribution support”](#) describes the support for distribution of basic software (BSW) on multiple cores. The support is described along with typical use cases for BSW distribution.

6.8.2. Background information

This section provides background information on the multi-core support in AUTOSAR and ACG on the following topics:

- ▶ [Section 6.8.2.1, “Multi-core approach in AUTOSAR”](#) presents the multi-core approach in AUTOSAR.
- ▶ [Section 6.8.2.2, “Considerations when designing multi-core systems in AUTOSAR”](#) describes use cases of the multi-core and scheduling dependencies.

6.8.2.1. Multi-core approach in AUTOSAR

In AUTOSAR, application software is distributed to the cores of the processor statically at integration time. This core assignment is done on the granularity level of OS-applications (see [\[12\]](#) and [\[13\]](#)). An OS-application encompasses a collection of operating system objects (e.g. tasks, ISRs, hook functions and alarms) and is further associated with a set of memory access rights.

In this core assignment, an OS-application that implements an `EcucPartition` is mapped to exactly one core. This means that OS-applications cannot be spread across several cores.

Communication between OS-applications that are mapped to different cores is established via the `Rte` module which again relies on mechanisms of the underlying OS. Sender-receiver as well as client-server communication is supported.

For synchronization among software that is executed on different cores, interrupt locks and resources cannot be used because they are effective on a single core only. Therefore, AUTOSAR defines spinlocks that can be used to protect concurrent access by tasks and ISRs on different cores. A spinlock is a busy waiting synchronization mechanism that polls a defined memory location until the variable assumes a defined value. This busy waiting causes scheduling implications that are detailed in [Section 6.8.2.2, “Considerations when designing multi-core systems in AUTOSAR”](#).

Furthermore, AUTOSAR specifies the support for the distribution of BSW modules to different cores. This is referred to as *BSW distribution* (or formerly *Enhanced BSW allocation*). The basic concepts are described in [\[11\]](#). For distribution of BSW modules, two concepts exist, which are usually combined.

With the master-satellite approach, a single BSW module is distributed to multiple cores. While one instance, i.e. the master, usually performs the bulk of the processing, the satellites perform a certain functionality of the

BSW module on the local core. The implementation of the master-satellite approach is always module-specific and must be specifically supported in the module.

The second concept considers the distribution of complete BSW module instances to cores, i.e., each BSW module instance with its full functionality is assigned to a core. In this setup, the communication between BSW modules that are mapped to different cores is established via the BSW Scheduler module (`SchM`). Consequently, also the second concept must be specifically supported by the BSW modules as they must not call APIs of BSW modules on another core directly but via a `SchM` client-server call.

For further details about the BSW distribution support in EB tresos AutoCore Generic, see [Section 6.8.4, “BSW distribution support”](#).

6.8.2.2. Considerations when designing multi-core systems in AUTOSAR

Multi-core systems are typically used for additional processing power. However, using a processor with two cores does not mean that the programs to be executed run twice as fast.

As explained in [14], the speed-up of a program which uses multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program. Therefore, the execution of application code that cannot be parallelized on a multi-core processor can indeed decrease performance because of the synchronization overhead between the cores. This should be taken into account during the design of a multi-core system.

Especially the introduction of execution dependencies between cores should be avoided. The prime sources for such dependencies, which are typically introduced at integration time, are:

- ▶ Multi-core locks (i.e. spinlocks) because they can block further execution on one core until the other core releases the lock depending on task priorities.
- ▶ Synchronous calls between cores because the calling application cannot continue processing until the called core schedules the request and returns the result.

An example of the scheduling implications of a spinlock is illustrated in [Figure 6.46, “Multi-core scheduling with spinlock”](#). The implications of synchronous calls between cores are comparable and therefore no separate example is provided.

In the example, Task 1 and Task 3 share an exclusive area that is protected using a spinlock. Task 3 tries to obtain the spinlock, which is already held by Task 1 on the other core. Task 3 now performs busy waiting on the spinlock, thereby preventing Task 2 from executing. Only after Task 1 releases the spinlock can Task 3 continue processing. In this example, Task 2 could have completed earlier if Task 3 had not been busy waiting for the spinlock. Note that Task 2 has no functional dependency to Core 0 but is simply blocked from executing by the access of Task 3 to the spinlock.

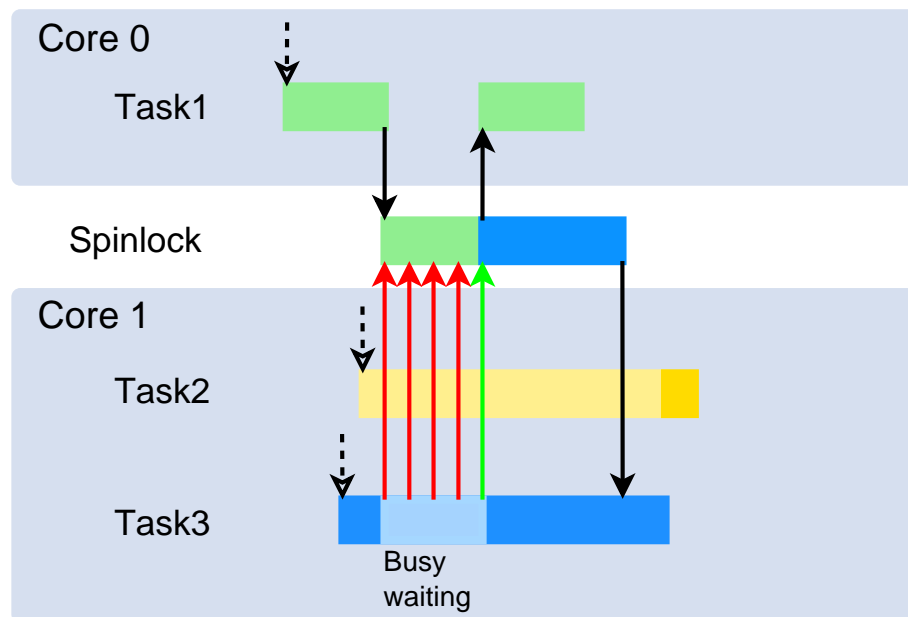


Figure 6.46. Multi-core scheduling with spinlock

As the example shows, complex scheduling dependencies can arise from the use of spinlocks and synchronous calls between cores. Furthermore, the overhead introduced is often not negligible because the operations may require system calls that depend on the underlying hardware platform.

Besides the temporal aspects in the design of a multi-core system, the memory mapping shall also be explicitly considered. A wide range of automotive multi-core processors feature a non-uniform memory architecture. In these architectures, certain address ranges are accessible fast from one core while the access is slow from another (i.e. scratchpad memories). Furthermore, if caches are used, they are often separate for each core, and no cache coherency protocol is in place. Thus, if caching is enabled, a core may read old data if the data at a given address was modified from another core and the caches were not explicitly flushed and invalidated.

In order to ensure fast memory access and consistent data, data should be allocated to memory sections that are specifically mapped for specific access patterns. For example, data only used by one core should be mapped to that core's scratchpad memory while data that is shared between multiple cores must be mapped to uncached memory regions (in case of non-coherent caches).

For application software, these considerations are part of the application design and the integration of the ECU. As for BSW modules that follow the master-satellite concept for BSW distribution, multiple memory sections may be defined specifically for each instance depending on the implementation.

6.8.3. Application support

As explained in [Section 6.8.2.1, “Multi-core approach in AUTOSAR”](#), an OS-application is considered to be a partition.

On single-core systems, OS-applications are only available for scalability classes 3 and 4 because OS-applications are only used for memory protection. In contrast, on multi-core systems, OS-applications are available independently from the operating system scalability class. This is because all OS objects (tasks, ISRs, etc.) must belong to an OS-application (see [Section 6.8.2.1, “Multi-core approach in AUTOSAR”](#)).

Additionally, it can be necessary to define a set of memory regions that can be accessible from an OS-application (for memory protection). Therefore, an OS-application should reference a set of memory regions which defines the access rights on certain memory ranges.

Furthermore, ACG provides a set of features specifically targeted at improving multi-core use cases that are described in the following sections.

- ▶ [Section 6.8.3.1, “Communication between cores”](#) explains options for communication between cores.
- ▶ [Section 6.8.3.2, “Locking”](#) explains how to use efficient locking mechanisms to protect exclusive areas.
- ▶ [Section 6.8.3.3, “Result-free client server communication”](#) explains how to use result-free asynchronous client-server communication.

6.8.3.1. Communication between cores

ACG8 RTE offers two different options for communication between cores:

- ▶ Shared memory communication
- ▶ Inter OS-application communicator (IOC).

Both options target specific use cases for communication between cores. Therefore the preferred option depends on the application.

Shared memory communication establishes the communication by joint access of the communication partners to the same memory. The RTE module decides on the exact number of buffers depending on the core assignment and scheduling parameters of the communication endpoints. Due to the reduced buffering, only few or no copy operations are required for this communication pattern. However, locking may be required if concurrent access is not prevented by other means, as for example by runnable to task mapping, priority assignment or time-driven scheduling. Consequently, this communication mechanism is best suited when large amounts of data shall be transferred and concurrent access is prevented by other means.

On the other hand, the IOC establishes the communication via special data structures that do not require locking between the sender and receiver. Therefore, a one-to-one communication is established lock-free, and scheduling dependencies between the communication endpoints are not introduced.

If multiple writers send on the same channel, a lock must be configured to ensure exclusive access. Since sender and receiver are not synchronized, this type of communication requires buffering and consequently involves two copy operations of the data. As a result, this communication mechanism is best suited when no

information on the concurrency of the communication endpoints is available, e.g. communication channels within the BSW.

For detailed explanation of the shared memory communication and the IOC, see the ACG8 RTE user documentation.

6.8.3.2. Locking

ACG8 RTE offers multiple configuration options for the realization of the data consistency mechanism of exclusive areas. The different mechanisms apply to different scopes of locking, i.e., whether a lock is used internally within a partition, between partitions, or between cores. For configuration in an ECU project, the smallest applicable locking scope should be selected to improve performance. The exact implementation depends on the hardware platform used. For further details, see [Section 7.3.2, “Mapping exclusive areas in the basic software modules”](#).

6.8.3.3. Result-free client server communication

As stated in [Section 6.8.2.2, “Considerations when designing multi-core systems in AUTOSAR”](#), synchronous cross-core calls pose a big impediment to exploiting the full computation performance of multi-core architectures. Therefore, asynchronous communication should be preferred.

For some APIs defined as synchronous in AUTOSAR, the return value is typically not evaluated by the caller, e.g. `Dem_SetEventStatus()`. Therefore, an asynchronous implementation may provide performance benefits. However, according to AUTOSAR, the result of an asynchronous operation shall be fetched using the `Rte_Result()` API before executing the server again. In the case of APIs for which the return value is never evaluated, this is unnecessary.

In order to improve performance, ACG8 RTE supports result-free asynchronous client-server calls where the caller can ignore the return result (RFC #70295). [Figure 6.47, “Result-free asynchronous client-server call”](#) illustrates this option. A SWC may call the API multiple times without any call to the `Rte_Result()` API, thus avoiding the necessity for an additional cross-core communication call. However, the previous call must be complete before the next is issued. The result-free asynchronous client-server calls therefore provide an easy way to dispatch calls to another core and proceed with local processing. When the underlying communication channels are realized via the IOC as described in [Section 6.8.3.1, “Communication between cores”](#), no scheduling dependencies between the cores are introduced.

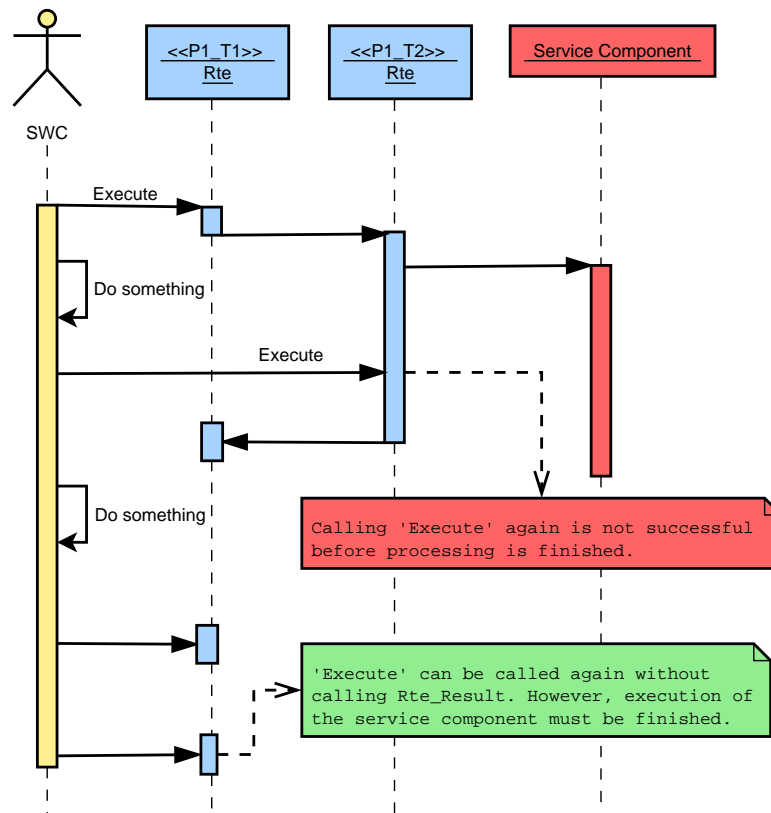


Figure 6.47. Result-free asynchronous client-server call

6.8.4. BSW distribution support

In this section, the basic concepts behind the BSW distribution are explained. Following this, a short overview of the support for BSW distribution in EB tresos AutoCore Generic products is provided. For more specific information, see the related product user documentation.

NOTE



License protected feature

For some EB tresos AutoCore Generic products, the BSW distribution feature is license protected. This information can be found in the product-specific user documentation supported features list.

6.8.4.1. Concept

As described in [Section 6.8.2, “Background information”](#), AUTOSAR permits you to map BSW modules to multiple partitions and therefore across multiple cores using BSW distribution. As mentioned earlier, the implementation is module-specific and can support one of the following options:

- ▶ Instances of the module can be completely mapped to several cores. The communication with the remainder of the BSW is established using the `SchM`.
- ▶ A module can be distributed across cores in a master-satellite implementation that executes parts of its functionality on another core.

These options are described in further detail in the following sections.

6.8.4.1.1. Mapping instances of BSW modules to multiple partitions/cores

When a complete module is mapped to a different partition, the `SchM` is responsible for API execution, i.e., API calls must be realized via `SchM_Call()` operations rather than direct function calls. Therefore, the Basic Software Module Description (BSWMD) of the modules defines the module's inter-partition interfaces. In the example in [Figure 6.48, “Mapping instances of BSW modules to multiple partitions/cores”](#), interfaces of the `EcuM` module are defined in the BSWMD so that the `SchM` can generate the appropriate functions for inter-partition communication, both for communication on one core or for communication between multiple cores.

Additionally, the same BSW module can be fully instantiated for different partitions. In this case, the same interfaces are available on all cores independently. As a result, each instance of the BSW module has core-specific configuration parameters (both flash and RAM), and can also have different behavior depending on the core it is mapped to. The specific behavior is determined at runtime using the operating system APIs, e.g. `GetCoreID()` or `GetApplicationID()`.

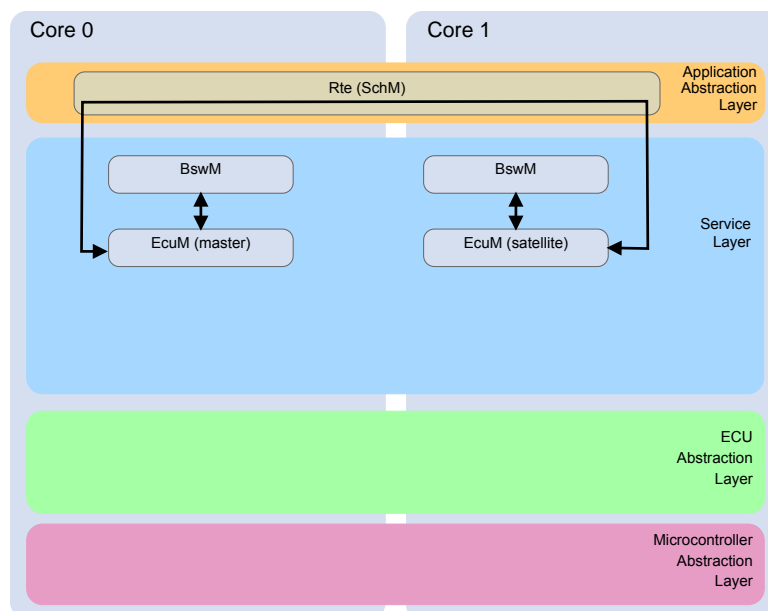


Figure 6.48. Mapping instances of BSW modules to multiple partitions/cores

The `SchM_Call()` mechanism is implemented in the following modules:

- ▶ `Dcm`, as described in chapter *BSW distribution* of the EB tresos AutoCore Generic 8 Diagnostic Stack documentation.
- ▶ mode management modules `Nm`, `ComM`, `EcuM`, and `BswM`, as described in [Section 6.8.4.2, “BSW distribution support in mode management”](#) and chapter *BSW distribution* in the EB tresos AutoCore Generic 8 Mode Management documentation.

6.8.4.1.2. Distribution of master-satellite BSW modules to multiple partitions/cores

To enable higher concurrency of functionality, BSW modules can be distributed across different partitions using the master-satellite approach:

- ▶ As implementation of synchronization between master and satellites is implementation-specific, the master and satellite of a BSW module must be provided by the same vendor.
- ▶ As described in [Section 6.8.4.1.1, “Mapping instances of BSW modules to multiple partitions/cores”](#), the functionality of the BSW using the master-satellite approach can be branched depending on the partition on which the code is executed. This can be done for example by using `GetCoreID()` or `GetApplicationID()`. Other implementations are also possible, and each core can execute its own code.
- ▶ Distributed BSW modules provide a `BswModuleImplementation` per instance that must be mapped to an OS-application in the `Rte` configuration.

The `Dem` module uses the master-satellite approach. This allows other distributed BSW modules to report diagnostic events to the `Dem` regardless of their location. For details, see the EB tresos AutoCore Generic 8 Diagnostic Stack documentation, chapter *Support for BSW event reporting from multiple cores*.

6.8.4.2. BSW distribution support in mode management

EB tresos AutoCore Generic supports BSW distribution for the mode management stack as specified by AUTOSAR in [\[15\]](#) and [\[16\]](#). The goal of the BSW distribution is to establish synchronized startup, sleep, wakeup and shutdown sequences across multiple cores.

The BSW distribution of the mode management is realized by processing the bus-specific activities on the core where the communication stack is located. The bus-independent activities are processed on the master core.

The BSW distribution architecture of ACG8 Mode Management is depicted in [Figure 6.49, “Distribution of mode management”](#) and is composed of

- ▶ one `EcuM` instance per core
- ▶ one `BswM` instance per partition
- ▶ one `ComM` master per core and one `ComM` satellite per core if partial networking is used

- ▶ one Nm on a master core

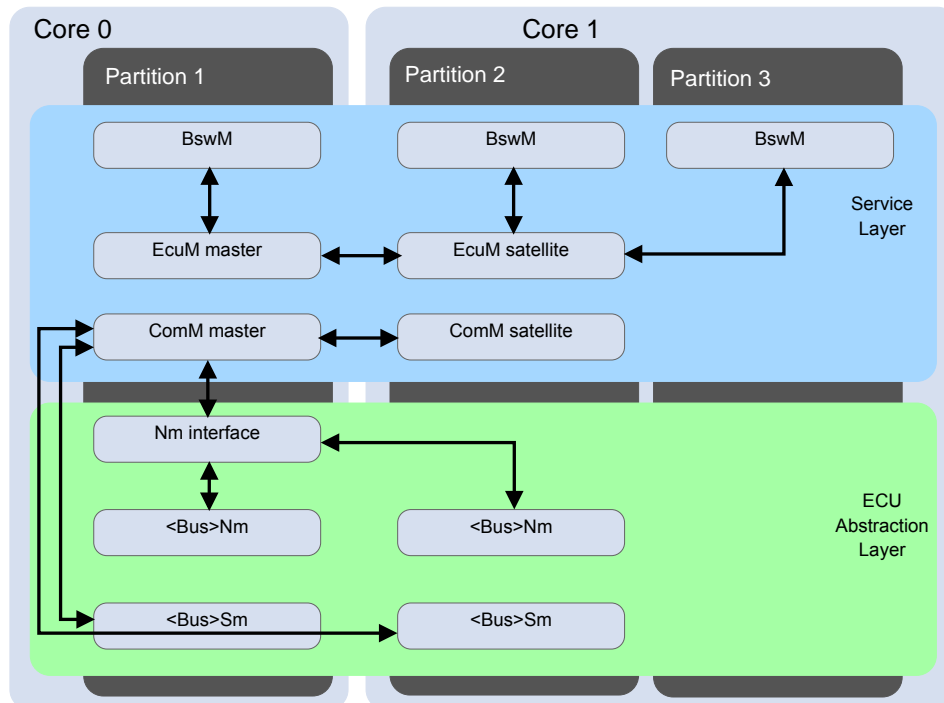


Figure 6.49. Distribution of mode management

The `EcuM` is distributed based on a master-satellite approach where the master is responsible for synchronizing the ECU states of all `EcuM` satellites across the different cores. The `BswM` uses multiple instantiation, i.e. multiple identical instances, which only communicate with their core-specific `EcuM`. If modes of the separate `BswM` instances shall be synchronized across cores, `BswMModeRequestPorts` must be configured for the instances. Communication between the `EcuM` master and satellite is realized by calling `SchM` interfaces.

The ComM provides one master that is mapped to the same core as the Nm. This ComM master processes both channel and PNC state machines. The ComM uses direct function calls when calling the Nm because ComM and Nm are always mapped to the same core. The ComM uses inter-core calls via SchM for calling the bus-specific state management modules. The Nm uses inter-core calls via SchM for calling the bus-specific network management modules.

If partial networking is used in the project, the ComM provides satellites that process the partial network Com signals on the core where the relevant PDUs are mapped. Only transmission and reception of the Com signals are processed locally on the core. Then, via SchM inter-core communication, the PNC information is exchanged with the master core where the processing is done.

Diagnostic requests for activating/deactivating the communication are also handled by inter-core calls using SchM.

For detailed information on the support for BSW distribution of mode management, see the ACG8 Mode Management user documentation.

6.8.4.3. BSW distribution of diagnostic log and trace

EB tresos AutoCore Generic supports BSW distribution in the `Dlt` module in order to provide increased performance of `Dlt_SendLogMessage()` and `Dlt_SendTraceMessage()` in a multi-core project.

Typically, the APIs `Dlt_SendLogMessage()` and `Dlt_SendTraceMessage()` are called frequently from SWCs and BSW modules, respectively. When the callers are mapped to a core that does not have the `Dlt` module, and without BSW distribution, each call results in a synchronous call between cores. Depending on the current configuration, `Dlt` then applies log level filtering, and therefore neglects a majority of the calls received.

The BSW distribution of the `Dlt` establishes this log level filtering directly in the satellite instances. This avoids communication between cores whenever the log messages are subject to filtering.

In order to establish this use case, the `Dlt` master must be mapped to the same partition as the `PduR` and the `NvM`. This is because the `Dlt` module handles the external communication and persistent storage of log messages in non-volatile memory. Furthermore, the satellite instances must be mapped to each partition from which `Dlt_SendLogMessage()` and `Dlt_SendTraceMessage()` shall be called.

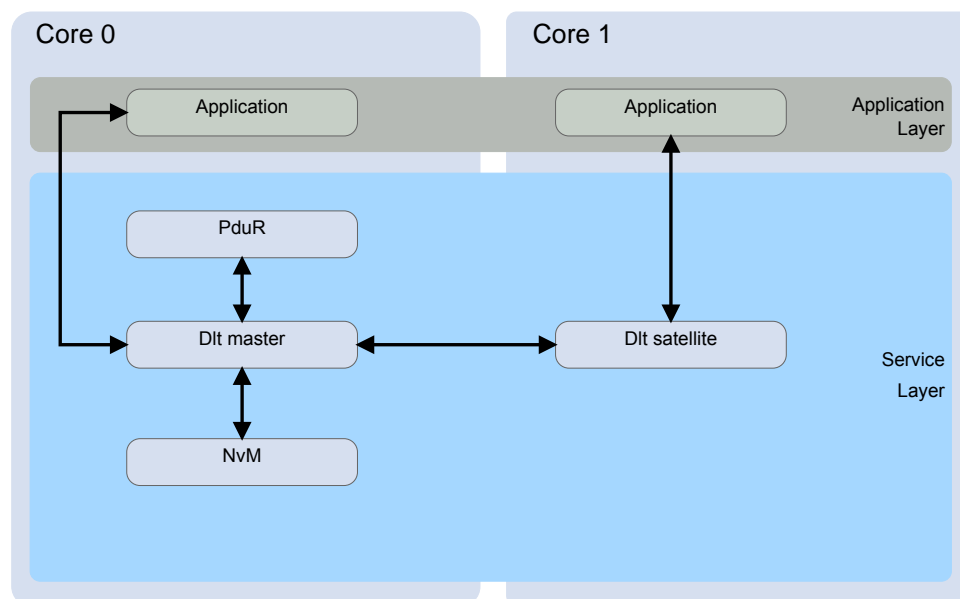


Figure 6.50. Distribution of diagnostic log and trace

For detailed information on the support for BSW distribution of `Dlt`, see the ACG8 DLT user documentation.

7. Integration

7.1. Overview

This chapter covers a number of aspects that are important for the integration task:

- ▶ [Section 7.2, “Integration first steps”](#) describes the first steps to take when you start to integrate a project.
- ▶ [Section 7.3, “Integration notes”](#) describes some general concepts and instructions about exclusive areas, production errors, memory mapping, and compiler abstraction .
- ▶ [Section 7.4, “Optimizing EB tresos AutoCore”](#) provides information on ways to optimize your project both at module level and then in the later stages of the integration process.

7.2. Integration first steps

This chapter will enable you to understand the basic concepts of the AUTOSAR integration issues.

In order to begin with the task of integration, the following topics have to be considered:

- ▶ task design - the tasks that are needed for the running ECU
- ▶ integrating the software components (SWCs) - the effect of ECU states on a SWC
- ▶ ECU state handling - the role of the `ECuM` and `BswM` modules

7.2.1. Recommended task design

Before you are able to integrate the basic software stack, you need to define the tasks within an ECU. This chapter defines a task design which allows the integration of an AUTOSAR basic software stack, including the watchdog stack. While there are other possibilities to get the stack running, the following is the recommended way of doing it.

To be able to distinguish between the different usage of OS tasks, the list below provides an overview of the different kinds of OS tasks needed:

1. OS tasks for the SWCs managed by the `Rte`. See the EB tresos AutoCore Generic RTE documentation for details on configuring these.
2. OS tasks for the BSW main functions managed by the BSW Scheduler (`SchM`) in the `Rte`. See the EB tresos AutoCore Generic RTE documentation for details on configuring these.
3. OS task for the watchdog stack. See [Section 7.2.1.1, “Watchdog task example”](#) for details.

4. `Init` task for managing the startup phase. This task must be configured in the `Os` so that it is automatically started.

7.2.1.1. Watchdog task example

The OS task e.g. `WatchdogTask` has a high priority within all ECU tasks. The priority chosen ensures that the watchdog manager is activated often enough and therefore keeps the ECU alive. Within the OS task, the `WdgM` main function is called as shown in the following example:

```
TASK(WatchdogTask)
{
    WdgM_MainFunction();
}
```

7.2.2. Software component integration

The states of the basic software stack have an impact on the software component (SWC) design. For example, before shutting down an ECU, the SWCs have to be notified to write their non-volatile (NV) data to NV memory using the memory stack. The SWCs are notified about such mode switches using the mode switch events. See the EB tresos AutoCore Generic RTE documentation for further details about mode switch events.

7.2.3. ECU state handling

The state handling of the basic software stack is managed by the `ECU State Manager` module (`EcuM`) and the `BSW Mode Manager` module (`BswM`). The `EcuM` and the `BswM` are responsible for system initialization, system shutdown and overall ECU state management.

The `ECU State Manager` and the `BSW Mode Manager` are part of the mode management product. You find further information about these modules and their setup in the EB tresos AutoCore Generic Mode Management documentation.

7.3. Integration notes

This section describes some general concepts that you need to be aware of during integration:

- ▶ [Section 7.3.1, “Deviations from MISRA rules”](#) provides information on MISRA rule deviations, their risks, and how to avoid problems related to these MISRA rule deviations.

- ▶ [Section 7.3.2, “Mapping exclusive areas in the basic software modules”](#) provides information about the concept of exclusive areas in the basic software together with recommended mappings for these.
- ▶ [Section 7.3.3, “Service Needs Calculator”](#) This chapter provides general information about the `Service Needs Calculator`
- ▶ [Section 7.3.4, “Production errors”](#) introduces the concept of production errors as defined by AUTOSAR and how these may be configured using the `Service Needs Calculator`.
- ▶ [Section 7.3.5, “Memory mapping and compiler abstraction”](#) describes the idea behind memory mapping and compiler construction and how you can configure these.

For detailed module related information about each of the topics above, see the module references chapter of the product-specific documentation.

7.3.1. Deviations from MISRA rules

EB tests the EB tresos AutoCore against MISRA rules. In certain cases, EB tresos AutoCore deviates from MISRA rules. This chapter provides information on how to handle these deviations.

For further information on MISRA deviations, contact your Elektrobit Automotive GmbH representative.

MISRA deviation	Description
Identifier character length	According to MISRA rules, an identifier needs to have at least 32 significant characters. Significant characters are characters that make an identifier unique. However, depending on your EB tresos AutoCore software configuration, an identifier can have more than 32 characters. Therefore, you need to make sure to use a compiler or linker that supports as many significant characters as are used in the longest identifier. Consider also the identifiers from generated code.

7.3.2. Mapping exclusive areas in the basic software modules

7.3.2.1. Overview

The following sections provide basic information about mapping exclusive areas in the basic software modules. Further information can be found in the EB tresos AutoCore Generic RTE documentation.

- ▶ [Section 7.3.2.2, “Background information”](#) describes some basic use-cases to guide your choice for configuring the exclusive areas.
- ▶ [Section 7.3.2.3, “Configuring the exclusive areas”](#) provides information on where the exclusive areas are configured.

7.3.2.2. Background information

This section describes the different mapping mechanisms that you can use to protect exclusive areas, also known as critical sections, in the basic software.

NOTE**Prior knowledge of OS concepts is required**

The following descriptions assume that you have prior knowledge of OS concepts. When you choose a locking mechanism, you have to consider details of processor privileges, interrupt categories, interrupt locking mechanisms, use of resources and task scheduling with priorities.

The following options are available for mapping the exclusive areas:

► All Interrupt Blocking

This is implemented using the standard `Os` API functions `SuspendAllInterrupts/ResumeAllInterrupts`.

You can always use this locking mechanism.

This is the recommended setting unless:

- you wish to use category 1 interrupts to perform some actions independently of the basic software. In this case, refer to `OS Interrupt Blocking` instead.
- you wish to improve the run-time performance. In this case, refer to `EB Fast Lock or Disable Exclusive Area` instead.

► OS Resource

This is implemented using the standard `Os` API functions `GetResource/ReleaseResource`.

To use this locking mechanism, you must configure all tasks, hook functions and interrupts that can access the exclusive area, directly or indirectly, to share the same resource using the configuration parameter `OsTaskResourceRef`.

WARNING**Do not use resources to protect exclusive areas in the basic software modules**

Setting up the correct resource handling is complex and error-prone as this requires in-depth knowledge of all basic software modules and their interactions.

Since these interactions are configuration dependent, you have to analyze both the initial configuration and any change of configuration to ensure that critical sections will be protected as you expect. This is known to be problematic in practice.

Do not use resources to protect exclusive areas in the basic software modules.

► Cooperative Runnable Placement

Do not use this locking mechanism. This is intended for use with software components only.

► OS Interrupt Blocking

This is implemented using the standard `Os` API functions `SuspendOSInterrupts/ResumeOSInterrupts`.

Use this locking mechanism to protect the critical section from interruption due to category 2 interrupts or task switching to a higher priority task.

To use this locking mechanism, you must ensure that all interrupt service routines in the basic software modules that directly or indirectly invoke functions that access the exclusive area are triggered by category 2 interrupts only. This is the expected case and is essential if you use the `OS interrupt Blocking` mapping.

Note that this interrupt locking mechanism locks category 2 interrupts but not category 1 interrupts. Choose this locking mechanism instead of `All Interrupt Blocking` if you want to perform actions that are independent of the basic software in the context of a category 1 interrupt.

► EB Fast Lock

This is implemented using direct assembler instructions.

To use this locking mechanism in order to disable all interrupts (category 1 and 2) without using the standard `Os` API functions, you must execute the EB tresos AutoCore in supervisor mode. In this case, do not use partitioning support in the `Rte`.

► Disable Exclusive Area

Use this mechanism if conditions specified in the module references are met. Refer to the module references for your module, section `Exclusive Areas in the Integration notes` for details.

7.3.2.3. Configuring the exclusive areas

Configure the exclusive areas for the basic software modules using the `Rte Editor` in the `Rte` module.

7.3.3. Service Needs Calculator

7.3.3.1. Overview

This chapter provides general information about the unattended wizard `Service Needs Calculator`. If you need specific information about the `Service Needs Calculator` in relation to a certain modules, products, or on how to use in EB tresos Studio, see in the related documents that are delivered with the software.

- ▶ [Section 7.3.3.2, “Background information”](#) describes basic concepts of the `Service Needs Calculator`.
- ▶ [Section 7.3.3.3, “Generic and specific services”](#) describes the difference between a generic and a specific request.
- ▶ [Section 7.3.3.4, “Mainfunction period”](#) describes the mainfunction period.
- ▶ [Section 7.3.3.5, “Dem events”](#) describes the usage of `Dem` events.
- ▶ [Section 7.3.3.6, “Nvm block”](#) describes the usage of `NvM` blocks.
- ▶ [Section 7.3.3.7, “Init function”](#) describes the usage of `Init` functions.
- ▶ [Section 7.3.3.8, “BSW components and their Service Needs”](#) shows the entire overview of software components.
- ▶ [Section 7.3.3.9, “Functional decomposition”](#) describes the software components and the functionality of the independent parties of the `Service Needs Calculator`.

7.3.3.2. Background information

In AUTOSAR, a service is a logical entity that offers general functionality to software components and BSW modules. For example, the `NvM` offers services to read and write data blocks to the memory and is regarded as a service provider. To use such a service, a software component or BSW module may have a configuration dependency on the service provider.

For example, to use the `NvM` services, memory blocks must be configured in the `NvM`. References to these blocks must be configured in the software component itself or in the BSW module that wants to use the `NvM` services. This kind of module dependency is known in AUTOSAR as Service Needs.

In EB tresos AutoCore, the concept of Service Needs is extended to include further dependencies that may not have a direct reference in the service requester.

For example, the `init` function of BSW modules must be configured in the `EcuM` module. But the BSW modules themselves do not need direct `EcuM` references for this. This kind of dependency is also treated as Service Needs between the BSW modules and the `EcuM`.

Service Needs can be seen as a contract between modules that provide services and modules that request services. This contract between service providers and service requesters is communicated by exchanging XML fragments that describe the requested Service Needs. The contract contains information on the configuration dependencies between BSW modules. The `Service Needs Calculator` coordinates and provides the lifecycle and graphical user interface.

The `Service Needs Calculator` is an unattended wizard. The `Service Needs Calculator` configures parameters that one BSW module needs in the configuration of another BSW module. These parameters configure a service that a module can offer to other modules. This works the same way as calling a `main` function.

The `Service Needs Calculator` automatically collects the requests from the service requester and performs the necessary configuration changes in the service provider.

The `Service Needs Calculator` is supplied with the EB tresos AutoCore and provides support for the automatic configuration of the following Service Needs:

- ▶ EcuM init functions
- ▶ Dem events
- ▶ NvM blocks
- ▶ Os Tasks
- ▶ Os ISRs
- ▶ Os resources
- ▶ Os schedule tables
- ▶ Os events
- ▶ Os IOC channels
- ▶ Os spinlocks
- ▶ Os alarms
- ▶ Com signals
- ▶ Com signal groups
- ▶ LdCom
- ▶ Xfrm Transformers (ComXf, E2EXf, SomelpXf)

7.3.3.3. Generic and specific services

A generic service is a predefined service for a use case that is needed by multiple modules. The `Service Needs Calculator` implements a set of services and provides a generic interface. The request for the generic service is done within a BSW module via the `plugin.xml` file. The request is parametrized via an XML file in the tresos plugin of the module. The execution of a generic service request depends on conditions, e.g. the configured main function interval in the ECU configuration must be greater than 0. If such a condition is not fulfilled, the request cannot be performed by the `Service Needs Calculator`.

Specific request: A module needs a special service and no generic implementation is available. In this situation, the Java generator of the BSW module directly accesses the Java interface of the `Service Need Calculator`.

7.3.3.4. Mainfunction period

Preconditions

- ▶ The configuration parameters that specify the main function interval of a module, e.g. `EthIfMainFunctionPeriod`, must be set to your needed values.
- ▶ The `Rte` and `Os` module must be part of the project.

The `Service Needs Calculator` automatically adds a schedule table entry for the corresponding `main` function in the task as they are defined in the XML configuration for this service request. If the task does not exist, this request creates a new task. All software modules that are part of a functional cluster, e.g. communication stack, refer to the same task. Main functions are automatically sorted based on their defined successors and predecessors.

Always run the `Service Needs Calculator` after you updated a parameter in the ECU configuration that specifies the main function interval of a BSW module.

7.3.3.5. Dem events

Preconditions

- ▶ You must enable the configuration parameter that enables `Dem` reports to the respective modules, e.g. `EthSMDemCtrlTestResultReportToDem`. These parameters are EB specific and allow to report a `Dem` event as Development error instead or completely disabling the report function.
- ▶ The `Dem` module must be part of the project.

The `Service Needs Calculator` automatically adds a `DemEventParameter` configuration in the `Dem` module, including the debounce algorithm.

Always run the `Service Needs Calculator` if you updated a parameter in the ECU configuration that enables or disables the reporting of a `Dem` event of a module.

7.3.3.6. Nvm block

Preconditions

- ▶ The respective feature in a BSW module that requires the usage of a memory block is enabled.
- ▶ The `NvM` module must be part of the project.

The `Service Needs Calculator` automatically adds a respective `NvMBlockDescriptor` configuration in the `NvM` according to the needs of the BSW module.

Always run the `Service Needs Calculator` if you enable or disable a functionality in a BSW module that needs storage in non-volatile memory.

Always run the `Service Needs Calculator` if your update of the ECU configuration impacts the required size in the non-volatile memory, e.g. the required size of `ComM` depends on the number of the configured `ComM` channels.

7.3.3.7. Init function

Preconditions

- ▶ The `EcuM` module must be part of the project.

The `Service Needs Calculator` automatically adds the respective `Init` function of the BSW module to the right `Init` list, which is `EcuM` dependent on its successors.

Always run the `Service Needs Calculator` if you add the `Dem` and/or the `Det` module to the ECU configuration.

7.3.3.8. BSW components and their Service Needs

The following table lists the Service Needs of the BSW modules that are part of Base:

Module	Service Needs
Xfrm	XFRMIMPLEMENTATIONMAPPING

Table 7.1. ACG Base

The following table lists the Service Needs of the BSW modules that are part of the Communication Stack:

Module	Service Needs
CanNm	MAINFUNCTION
CanSM	MAINFUNCTION and DEMEVENETS
CanTp	MAINFUNCTION
Com	MAINFUNCTION and EXCLUSIVEAREA
DolP	MAINFUNCTION
EthIf	MAINFUNCTION
EthSM	MAINFUNCTION and DEMEVENETS
EthSwt	MAINFUNCTION and DEMEVENETS
Fr	DEMEVENETS
FrIf	MAINFUNCTION and DEMEVENETS
FrNm	MAINFUNCTION

Module	Service Needs
FrSM	MAINFUNCTION and DEMEVENETS
FrTp	MAINFUNCTION
IpduM	MAINFUNCTION
LinIf	MAINFUNCTION and DEMEVENETS
LinSM	MAINFUNCTION
PduR	EXCLUSIVEAREA
Sd	MAINFUNCTION
SoAd	MAINFUNCTION
SomelpTp	MAINFUNCTION
Tcplp	MAINFUNCTION
Tftp	MAINFUNCTION
UdpNm	MAINFUNCTION

Table 7.2. Communication Stack

The following table lists the Service Needs of the BSW modules that are part of the Time Sync Stack:

Module	Service Needs
EthTSyn	MAINFUNCTION and DEMEVENETS
StbM	MAINFUNCTION and NVMBLOCK

Table 7.3. Time Sync

The following table lists the Service Needs of the BSW modules that are part of the Diagnostic Stack:

Module	Service Needs
Dcm	MAINFUNCTION and NVMBLOCK and INITFUNCTION
Dem	MAINFUNCTION and NVMBLOCK and INITFUNCTION
Dlt	NVMBLOCK
FiM	MAINFUNCTION

Table 7.4. Diagnostic Stack

The following table lists the Service Needs of the BSW modules that are part of the Memory Stack:

Module	Service Needs
Ea	MAINFUNCTION
Eep	MAINFUNCTION and DEMEVENETS

Module	Service Needs
Fee	MAINFUNCTION
NvM	MAINFUNCTION and DEMEVENETS

Table 7.5. Memory Stack

The following table lists the Service Needs of the BSW modules that are part of the Mode Management Stack:

Module	Service Needs
BswM	MAINFUNCTION
ComM	MAINFUNCTION and NVMBLOCK
EcuM	MAINFUNCTION and DEMEVENETS
Nm	MAINFUNCTION
Xcp	MAINFUNCTION and NVMBLOCK and DEMEVENETS

Table 7.6. Mode Management Stack

The following table lists the Service Needs of the BSW modules that are part of the Crypto and Security Stack:

Module	Service Needs
SecOC	MAINFUNCTION

Table 7.7. Crypto and Security Stack

The following table lists the Service Needs of the BSW modules that are part of the Watchdog Stack:

Module	Service Needs
WdgM	MAINFUNCTION and DEMEVENETS

Table 7.8. Watchdog Stack

7.3.3.9. Functional decomposition

All production errors can be configured in the `Dem` module manually. However, the `Service Needs Calculator` can automatically configure production errors for BSW modules. See the [EB tresos AutoCore Generic Base documentation](#) for detailed information about the `Service Needs Calculator`.




Configuring production errors with the `Service Needs Calculator`

Prerequisite:

- EB tresos Studio is installed and licensed.

Step 1

To open the `Service Needs Calculator` configuration dialog, click on the black arrow in the  button located in the menu bar of EB tresos Studio.

Step 2

In the **Unattended Wizards** list, click **Unattended wizard configuration....**

Step 3

Enable the **Calculate Service Needs** check box.

Step 4

In the **Active Service Needs for the current project** list, enable the **Dem Events** check box.

Step 5

To run the `Service Needs Calculator`, click .

7.3.4. Production errors

7.3.4.1. Overview

The following sections provide an introduction to production errors in the basic software modules.

- ▶ [Section 7.3.4.2, “Background information”](#) describes some basic information about production errors.
- ▶ [Section 7.3.4.3, “Configuring production errors”](#) provides information on how to configure production errors for the basic software modules.

7.3.4.2. Background information

The diagnostics event manager (DEM) is responsible for storing diagnostic events and associated information. These events are known as production errors or `Dem events`. Production errors are reported to the `Dem` module from software components (SWCs) or from basic software modules (BSWs).

You find information about the events that are reported to the `Dem` module from the BSWs, in the product-specific documentation. Refer to the module references for your module, section `Production errors` in the `Integration notes` for a list of the events that are reported.

7.3.4.3. Configuring production errors

All production errors can be configured in the `Dem` module manually. However, the `Service Needs Calculator` can automatically configure production errors for BSW modules. See the EB tresos AutoCore Generic Base product documentation for detailed information about the `Service Needs Calculator`. To configure production errors, select the option to automatically configure `Dem events`. Then run the `Service Needs Calculator`.

7.3.5. Memory mapping and compiler abstraction

7.3.5.1. Overview

The following chapter instructs you on how to configure the memory mapping and the compiler abstraction.

- ▶ To learn about the concept of memory mapping and compiler instruction, consult [Section 7.3.5.2, “Background information”](#).
- ▶ To configure the memory mapping or the compiler abstraction, follow instructions given in [Section 7.3.5.3, “Configuring the memory mapping and compiler abstraction”](#).

7.3.5.2. Background information

You may map all AUTOSAR basic software modules and all data to specific memory sections. This is called *memory mapping*.

To configure optimized (near) access to data and functions, use the compiler abstraction.

[Figure 7.1, “Overview of memory mapping and compiler abstraction”](#) shows the difference between the memory mapping and the compiler abstraction.

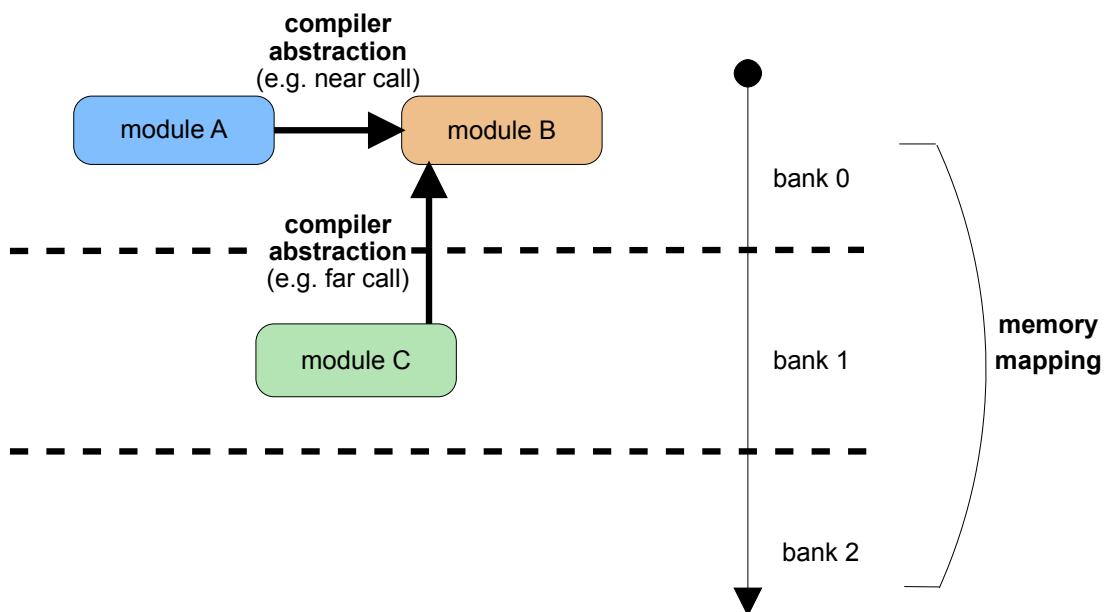


Figure 7.1. Overview of memory mapping and compiler abstraction

Memory mapping: module A and module B are *memory mapped* to bank 0. Module C is memory mapped to bank 1.

Compiler abstraction: As module A and module B are mapped to the same bank, the function calls from module A to module B are mapped to a near call by the *compiler abstraction*. As module C is located in another bank, the function calls from module C to module B must be mapped to far calls.

Besides the mapping of source code, the memory mapping allows to map data to specific sections. There are two common cases for the mapping of data.

1. The access to often used data is optimized if :
 - ▶ the data are mapped to the same bank as the code accessing the data (for 16-bit processors). This scenario is shown in figure [Figure 7.2, “Optimized memory usage with fast data access”](#)
 - ▶ the data are mapped to internal memory (in general).

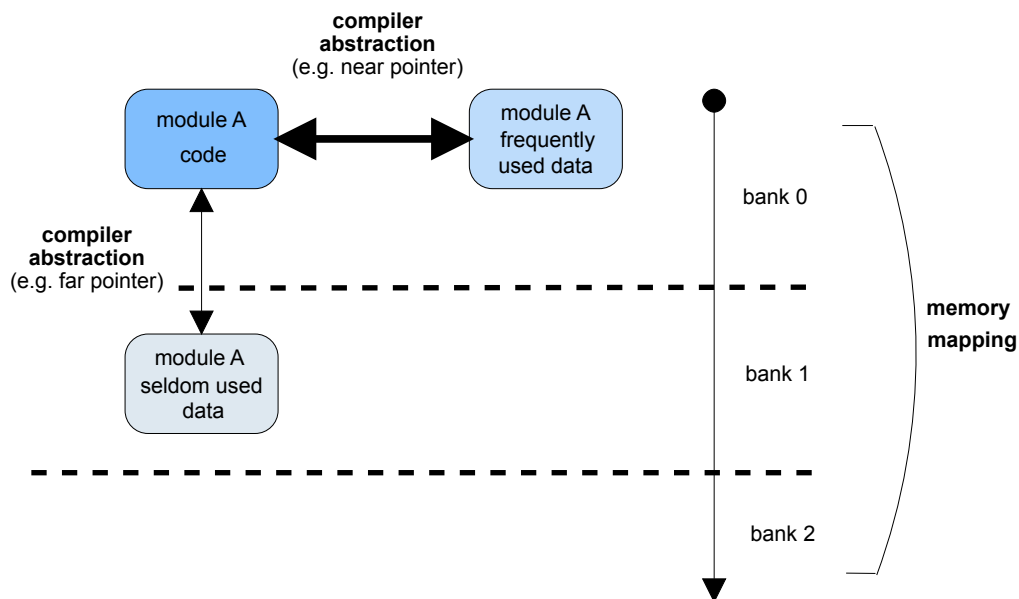


Figure 7.2. Optimized memory usage with fast data access

2. As data values are usually aligned according their size, you can avoid a waste of memory if you assign data with the same size to the same memory section. [Figure 7.3, “Optimized memory usage without gaps in memory”](#) shows how you can avoid gaps in the memory if you use different memory sections for data with different lengths.

All EB tresos AutoCore data are mapped according their size to specific memory sections.

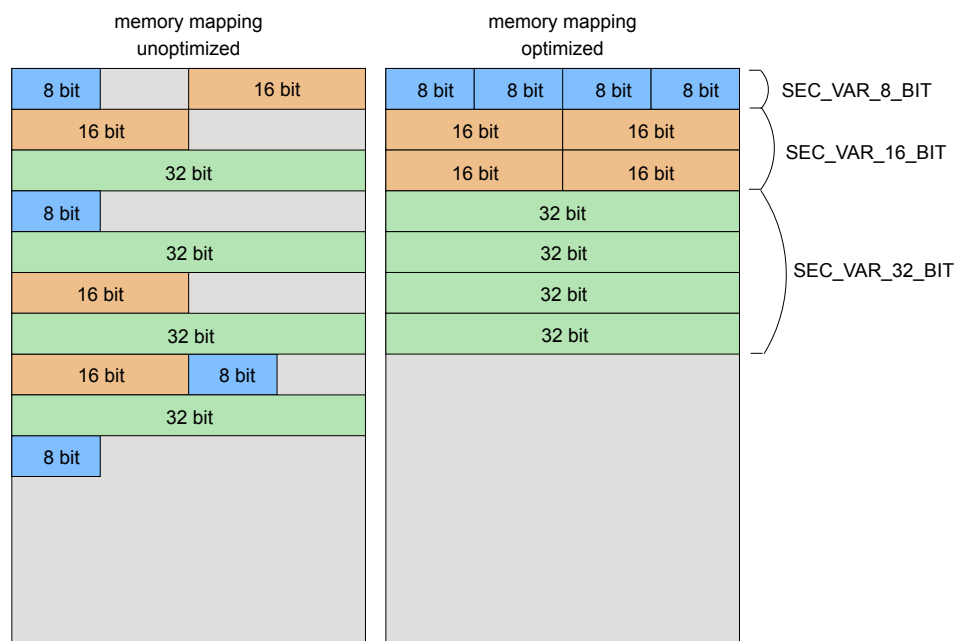


Figure 7.3. Optimized memory usage without gaps in memory

Find details about the configuration in [Section 7.3.5, “Memory mapping and compiler abstraction”](#)

7.3.5.3. Configuring the memory mapping and compiler abstraction

The following sections provide instructions about configuring the memory mapping and compiler abstraction. For memory mapping, follow the instructions in [Section 7.3.5.3.1, “Generating the memory mapping header files”](#). If you need to edit the memory mapping header file, follow the instructions in [Section 7.3.5.3.2, “Editing memory mapping header files”](#). To configure the compiler abstraction header file, follow the instructions given in [Section 7.3.5.3.3, “Editing the Compiler_Cfg.h file”](#).

7.3.5.3.1. Generating the memory mapping header files

The purpose of the memory mapping header files is to map the code and data sections, e.g. in the basic software (BSW) modules to specific linker sections.

The `MemMap` module generates the following files:

- `MemMap.h`: This file includes the `MemMapHeaderFiles` and the generated `<BSW>_MemMap.h`. The `MemMapHeaderFiles` is a list of header files configured by the user.

- ▶ `<SWC>_MemMap.h`: These files contain the memory allocation keywords for the `MemorySection` elements defined within the Software Component Description and compiler-specific instructions if valid `MemMap` mappings are created. One header file is generated for each instantiated software component type. `<SWC>` is the `shortName` of the software component type.
- ▶ `<BSW>_MemMap.h`: These files contain the memory allocation keywords for the `MemorySection` elements defined within the Basic Software Module Description and compiler-specific instructions if valid `MemMap` mappings are created. One header file is generated for each `BSW-Implementation`. `<BSW>` is composed of the `BswModuleDescription shortName`, the `vendorId` and the `vendorApiInfix` of the BSW module, appended by underscores. If no `vendorApiInfix` is defined for the BSW module, the `_vendorId_vendorApiInfix` is omitted from the name of the file.

NOTE



Common MemMap header file

The memory allocation keywords for `MemorySection` elements defined within the Basic Software Module Description and the Software Component Description are generated in the same header file if `MemMap` generates the same header file name for both the `BSW-Implementation` and the `SWC-Implementation`.

NOTE



You do not need to explicitly define section mappings for BSW modules

For most modules in the BSW, the information that is needed in the memory mapping header file can be generated with a default configuration. It is not necessary to adapt the mapping with compiler-specific instructions (pragma commands).



Generating memory mapping header files

Take the following steps to generate memory mapping header files.

Step 1

Add the `MemMap` module to your project.

Step 2

With the System Description Importer of EB tresos Studio, import the ARXML file from the configuration into the project.

Step 3

Generate the `swcd` ARXML files with `generate_swcd`.

Step 4

Run the wizard `Update Service Component and BSWM Descriptions`.

Step 5

If the configuration has Software Component Descriptions, configure the `RteSwComponentType` in the `Rte` module.

Step 6

Generate the code for your project.

The `MemorySection` elements are part of the SWS-Implementation and BSW-Implementation. The aggregation is realized by the element `ResourceConsumption`. The following attributes are mandatory for a `MemorySection`:

- ▶ `shortName`: the actual name of the memory section
- ▶ `swAddrMethod`: A reference to the common memory section. You can either refer to a `SwAddrMethod` defined in the BSW Module Description of the `Base` module, or you can define your own `SwAddrMethod` elements, depending on your use case.

The following attributes are optional for a `MemorySection` and are taken into account by the `MemMap` module:

- ▶ `alignment`: describes the alignment of objects within the `MemorySection`. The value should also be present in the `symbol` if the `symbol` is defined or in the `shortName` otherwise.
- ▶ `options`:
 - ▶ `coreScope` with the possible values `coreLocal` and `coreGlobal`. If `coreLocal` is set, the same option should be set in the referenced `swAddrMethod` and the word `LOCAL` should be present in the `symbol` if the `symbol` is defined or in the `shortName` otherwise.
 - ▶ `safety level` with the possible values `safetyQM`, `safetyAsilA`, `safetyAsilB`, `safetyAsilC` and `safetyAsilD`. If one of the ASIL safety options `safetyAsilA`, `safetyAsilB`, `safetyAsilC`, `safetyAsilD` is set, the same option should be set in the referenced `swAddrMethod`. Also, one of the strings `ASIL_A`, `ASIL_B`, `ASIL_C`, `ASIL_D` should be present in the `symbol` if the `symbol` is defined or in the `shortName` otherwise. Only one safety level option can be set. If no safety level keyword is added, the default shall be treated as `QM`, without any ASIL qualification.

The options `coreScope` and `safety level` are not mandatory. A `MemorySection` does not need to have the options defined but it can have both options defined or just one of them.

- ▶ `sectionNamePrefix`: used to set the memory section's namespace in the code. It is only used for BSW modules.
- ▶ `symbol`: used instead of the `shortName`.

Example of a `MemorySection`:

```
<!-- 8bit section -->
<MEMORY-SECTION>
  <SHORT-NAME>VAR_INIT</SHORT-NAME>
  <ALIGNMENT>8</ALIGNMENT>
  <OPTIONS>
    <OPTION>safetyAsilA</OPTION>
    <OPTION>coreLocal</OPTION>
  </OPTIONS>
  <SW-ADDRMETHOD-REF DEST="SW-ADDR-METHOD">
    /AUTOSAR_MemMap/SwAddrMethods/VAR_INIT_ASIL_A_LOCAL
  </SW-ADDRMETHOD-REF>
```



```
<SYMBOL>VAR_INIT_ASIL_A_LOCAL_8</SYMBOL>
</MEMORY-SECTION>
```

The pattern for the memory allocation keyword is `<PREFIX>_[START|STOP]_SEC_<NAME>` with the following parts:

`<PREFIX>`

For the BSW modules, this is composed of the `sectionNamePrefix` `shortName`, the `vendorId` and the `vendorApiInfix`, appended by underscores. If the `sectionNamePrefix` attribute is not defined, it is composed of the `BswModuleDescription` `shortName`, the `vendorId` and the `vendorApiInfix`, appended by underscores. If no `vendorApiInfix` is defined, the `_vendorId_vendorApiInfix` is omitted from the name of the file.

For the software components, it is the `shortName` of the software component type.

`<NAME>`

It is the `symbol` of the `MemorySection` if the `symbol` attribute is defined. If `symbol` is not defined, it is the `shortName` of the `MemorySection`.

NOTE



The ACG8 RTE does not support configurable mapping of runnable entities

The ACG8 RTE maps the function signature of runnable entities to the Rte-specific memory sections `RTE_START_SEC<PARTITION>_APPL_CODE` and `RTE_STOP_SEC<PARTITION>_APPL_CODE`. In this case, `<PARTITION>` is the name of the OS Application that the runnable entity is part of if partitioning is used. If you want to map runnable entities to particular memory regions, configure the affected memory section within the ECU configuration of the `MemMap` module.

If you want to define compiler-specific pragmas for each memory section, refer to the AUTOSAR specification of the `MemMap` module first [6]. The specification describes the configuration parameters and provides several examples on how to configure the module.

Refer to [Section 7.3.5.4, “Defining compiler-specific pragma commands”](#) for information that you must be aware of when you define compiler-specific pragmas.



Defining section-specific mappings

To define section-specific mappings between memory sections and compiler-specific pragmas, follow these steps.

Step 1

In EB tresos Studio, open the configuration of the `MemMap` module.

Step 2

Define a new `MemMapAddressingModeSet` and add a `MemMapAddressingMode` containing the pragmas you want to map.

Step 3

Define a new `MemMapAlignmentSelector` that is the same as the `alignment` attribute of the `MemorySection` you want to map.

Step 4

Define a new `MemMapAllocation` and add a `MemMapSectionSpecificMapping`.

Step 5

Within the `MemMapSectionSpecificMapping`, add a reference to the newly created `MemMapAddressingModeSet` and a reference to the memory section to which the pragmas shall be mapped. If the drop-down list for valid references to memory sections is empty, run the wizard `Update Service Component and BSWM Descriptions` first.



Defining section-generic mappings

To define section-generic mappings between memory sections and compiler-specific pragmas, follow these steps.

Step 1

In EB tresos Studio, open the configuration of the `MemMap` module.

Step 2

Define a new `MemMapAddressingModeSet` and add a `MemMapAddressingMode` containing the pragmas you want to map.

Step 3

Define one or more `MemMapAlignmentSelector` that are the same as the `alignment` attributes of the `MemorySections` you want to map.

Step 4

Define a new `MemMapAllocation` and add a `MemMapGenericMapping`.

Step 5

Within the `MemMapGenericMapping`, add a reference to the newly created `MemMapAddressingModeSet` and a reference to the `SwAddrMethod`. The pragmas shall be mapped to the `MemorySections` that reference this `SwAddrMethod`. If the drop-down list for valid `SwAddrMethod` references is empty, import the system description file where the `SwAddrMethod` elements are defined.

The compiler-specific preprocessor pragmas are set after the mapped `MemorySection`'s macros are undefined in the generated memory mapping header files.

NOTE



MemMapSectionSpecificMapping takes priority over MemMapGenericMapping

If a `MemMapGenericMapping` and a `MemMapSectionSpecificMapping` are configured for the same `MemorySection`, only the compiler-specific pragmas mapped via the `MemMapSectionSpecificMapping` are mapped to the `MemorySection`.

NOTE



SwAddrMethod filters

In a `MemMapAddressingModeSet`, attributes can be set to select what `SwAddrMethod` can be used in a valid `MemMapGenericMapping`:

- ▶ `MemMapSupportedAddressingMethodOption`: One of these attributes should match one of the options attribute of the `SwAddrMethod`.
 - ▶ `MemMapSupportedMemoryAllocationKeywordPolicy`: One of these attributes should match the `memoryAllocationKeywordPolicy` attribute of the `SwAddrMethod`.
 - ▶ `MemMapSupportedSectionInitializationPolicy`: One of these attributes should match the `sectionInitializationPolicy` attribute of the `SwAddrMethod`.
 - ▶ `MemMapSupportedSectionType`: One of these attributes should match the `sectionType` attribute of the `SwAddrMethod`.
-

The ACG8 RTE does not export dynamic `MemorySection` elements to its BSW Module Description. This affects `MemorySection` elements that contain the name of an OS Application and `MemorySection` elements for calibration parameters. You have to define the missing memory sections manually. Therefore you can configure additional header files that shall be included in the generated `MemMap.h`. This is described in the following chapter.

7.3.5.3.2. Editing memory mapping header files

For MCALs or the RTE, you must manually configure the memory mapping. To configure the memory mapping, you need to edit linker-specific header files and the linker-specific linker command file as described below.

You may put all mappings in one single header file. Alternatively, you may put the mappings into a number of different header files. You have to configure the file names of the header files in the `MemMap` module configuration using EB tresos Studio. The generated `MemMap.h` includes the configured header files.



Editing memory mapping header files

The code below shows an extract of a header file for the Greenhills toolchain:

```
#elif defined (CAN_AFCAN_START_SEC_PUBLIC_CODE)
#undef CAN_AFCAN_START_SEC_PUBLIC_CODE
#undef MEMMAP_ERROR
#pragma ghs section text=".CAN_PUBLIC_CODE_ROM"
```

Step 1

Map the `CAN_AFCAN_START_SEC_PUBLIC_CODE` section to the linker section `.CAN_PUBLIC_CODE_ROM`.

The section `CAN_AFCAN_START_SEC_PUBLIC_CODE` is defined in the basic software module `Can`.

Step 2

Map the linker section in the linker command file to a memory area as shown in the example:

```
SECTIONS
{
.CAN_PUBLIC_CODE_ROM      : > rom
...
}
```

Step 3

Assign the memory area to a physical address range in the linker command file. See the following code as an example:

```
MEMORY
{
rom : ORIGIN = 0x00000800, LENGTH = 0x000ff830 /*Internal ROM (1MB)*/
...
}
```

You finished your work in the memory mapping header file(s) and in the linker command file.

7.3.5.3.3. Editing the `Compiler_Cfg.h` file

To configure the compiler abstraction, you need to edit the compiler-specific `Compiler_Cfg.h` file.

The example below for the Metroworks compiler shows that memory class `FLS_CONST` is mapped to a near access and the pointer class `FLS_APPL_DATA` is mapped to a far access:

```
/** \brief definition of the constant memory class
**
** To be used for global or static constants. */
```

```
#define FLS_CONST __near

#if (defined FLS_APPL_DATA) /* to prevent double definition */
#error FLS_APPL_DATA already defined
#endif /* if (defined FLS_APPL_DATA) */

/** \brief definition of the application data pointer class
**
** To be used for references on application data (expected to
** be in RAM or ROM) passed via API. */
#define FLS_APPL_DATA __far
```

Compared to the example [Figure 7.2, “Optimized memory usage with fast data access”](#), the memory class `FLS_CONST` corresponds to the module A frequently used data and the pointer class `FLS_APPL_DATA` corresponds to module A seldom used data.

7.3.5.4. Defining compiler-specific pragma commands

Pragma commands can be defined to instruct the compiler about the location of code or data. These pragma commands are specific to toolchain and platform. You must consult the toolchain and platform documentation for information about using pragma commands.

This section provides an overview of the memory sections defined by AUTOSAR together with information needed when defining compiler-specific pragma commands for these sections. Full information about the AUTOSAR definitions and examples of pragma commands can be found in the AUTOSAR software specification for the MemMap module [\[6\]](#).

In general, the types of sections that are used by the EB BSW modules can be summarized by the following expression:

```
[
<INTERNAL_VAR|VAR|CONST>_[NO_INIT|INIT|POWER_ON_INIT]_<BOOLEAN|8|16|32|UNSPECIFIED>
|
CODE
]
```

The following examples give information needed when defining compiler-specific pragma commands for these types.

7.3.5.4.1. Examples

```
VAR_NO_INIT_16
```

The pragmas of the assigned MemMapAddressingMode definitions have to be set to a memory section that satisfies the following criteria:

- ▶ The memory section is located in RAM (VAR).
- ▶ The memory section is not initialized by the C startup code (NO_INIT).
- ▶ The start of the memory section is aligned to 16-bit boundaries (16).

CODE

The pragmas of the assigned MemMapAddressingMode definitions have to be set to a memory section that satisfies the following criteria:

- ▶ It is located in ROM.

VAR_INIT_32

The pragmas of the assigned MemMapAddressingMode definitions have to be set to a memory section that satisfies the following criteria:

- ▶ It is located in RAM (VAR).
- ▶ It is initialized by the C startup code upon every reset (INIT).
- ▶ The start of the memory section is aligned to 32-bit boundaries (32).

CONST_INIT_32

The pragmas of the assigned MemMapAddressingMode definitions have to be set to a memory section that satisfies the following criteria:

- ▶ It is located in ROM (CONST).
- ▶ The start of the memory section is aligned to 32-bit boundaries (32).

INTERNAL_VAR_POWER_ON_INIT_32

The pragmas of the assigned MemMapAddressingMode definitions have to be set to a memory section that satisfies the following criteria:

- ▶ It is located in RAM (INTERNAL_VAR).
- ▶ It is initialized by the C startup code upon every power on reset (POWER_ON_INIT).
- ▶ The start of the memory section is aligned to 32-bit boundaries (32).

VAR_INIT_BOOLEAN

The pragmas of the assigned MemMapAddressingMode definitions have to be set to a memory section that satisfies the following criteria:

- ▶ It is located in RAM (VAR).
- ▶ It is initialized by the C startup code upon every reset (INIT).

- ▶ The start of the memory section is aligned to the boundaries defined by the platform for the data type Boolean (BOOLEAN).

7.4. Optimizing EB tresos AutoCore

7.4.1. Overview

This chapter outlines the optimization of EB tresos AutoCore to improve your application's performance and reduce the code size of the basic software. Optimization can be considered at two main stages of development; module level, primarily focusing on individual modules, and project level, extending optimization to your project as a whole.

The types of optimization that can be applied are described in detail in [Section 7.4.2.1, “Types of optimization”](#). [Section 7.4.3, “Setting up a module for optimization”](#) describes the steps you can follow to start to optimize your modules. [Section 7.4.5, “Optimizing your project as a whole”](#) describes the steps you can follow to optimize your project further.

7.4.2. Background information

Optimization generally concerns two questions:

- ▶ my application is too big - how can I reduce it?
- ▶ my application is too slow - how do I make it run faster?

The first question concerns the consumption of RAM and ROM. At module level, this is affected by both the module configuration and the module implementation itself. Each module has a set of parameters that must be configured to suit your application. Some of these parameters can be used to influence the consumption of RAM and ROM and so can affect the optimization of the module in terms of its size. At project level, code size optimization can be achieved by disabling some common API functionality or by adjusting the implementation used for exclusive areas.

The second question concerns the implementation of the modules used and also, the way in which the modules are configured to run. At module level, the choice of configuration parameters can influence the implementation itself. At project level, disabling functionality, improving the scheduling and adjusting the management of exclusive areas can also lead to better performance.

In order to address the basic questions about optimization, different optimization types are defined. [Section 7.4.2.1, “Types of optimization”](#) describes these in further detail. Based on these optimization types, the options for optimizing your module and then your project are elaborated in [Section 7.4.3, “Setting up a module for optimization”](#) and [Section 7.4.5, “Optimizing your project as a whole”](#) respectively.

7.4.2.1. Types of optimization

Module specific optimizations affect both RAM and ROM consumption as well as execution time. We therefore consider the following types of optimization:

- ▶ ROM consumption of the configuration
- ▶ RAM consumption of the configuration
- ▶ ROM consumption of the static code
- ▶ RAM consumption of the static code
- ▶ execution time of the static code

The types of optimization listed above are used as a basis for configuring individual parameters at module level. Where possible, these types of optimization are documented in the parameter description. This information shows the impact of particular settings on the generated code. Refer to [Section 7.4.4, “Optimizing each parameter”](#) to see an example of this documentation.

Besides the module specific optimization types, some further optimizations are valid for your project as a whole. These are:

- ▶ turn off development error detection (Det) checks
- ▶ turn off the `GetVersionInfo()` API
- ▶ adjust the scheduling
- ▶ adjust the exclusive areas configuration

This method of optimization is described further in [Section 7.4.5, “Optimizing your project as a whole”](#).

7.4.3. Setting up a module for optimization

To set up each individual module for optimization, you may start with a recommended configuration when you create a new configuration project in EB tresos Studio.

When you select modules in the EB tresos Studio **Module Configurations** dialog, select a recommended configuration suitable for your intention.

Example: The `Com` module provides a number of recommended configurations. These provide a choice between a small, standard, medium and maximum configuration. Whereas a small configuration is suitable for simple ECUs, a maximum configuration provides more flexibility for complex ECUs like a gateway.

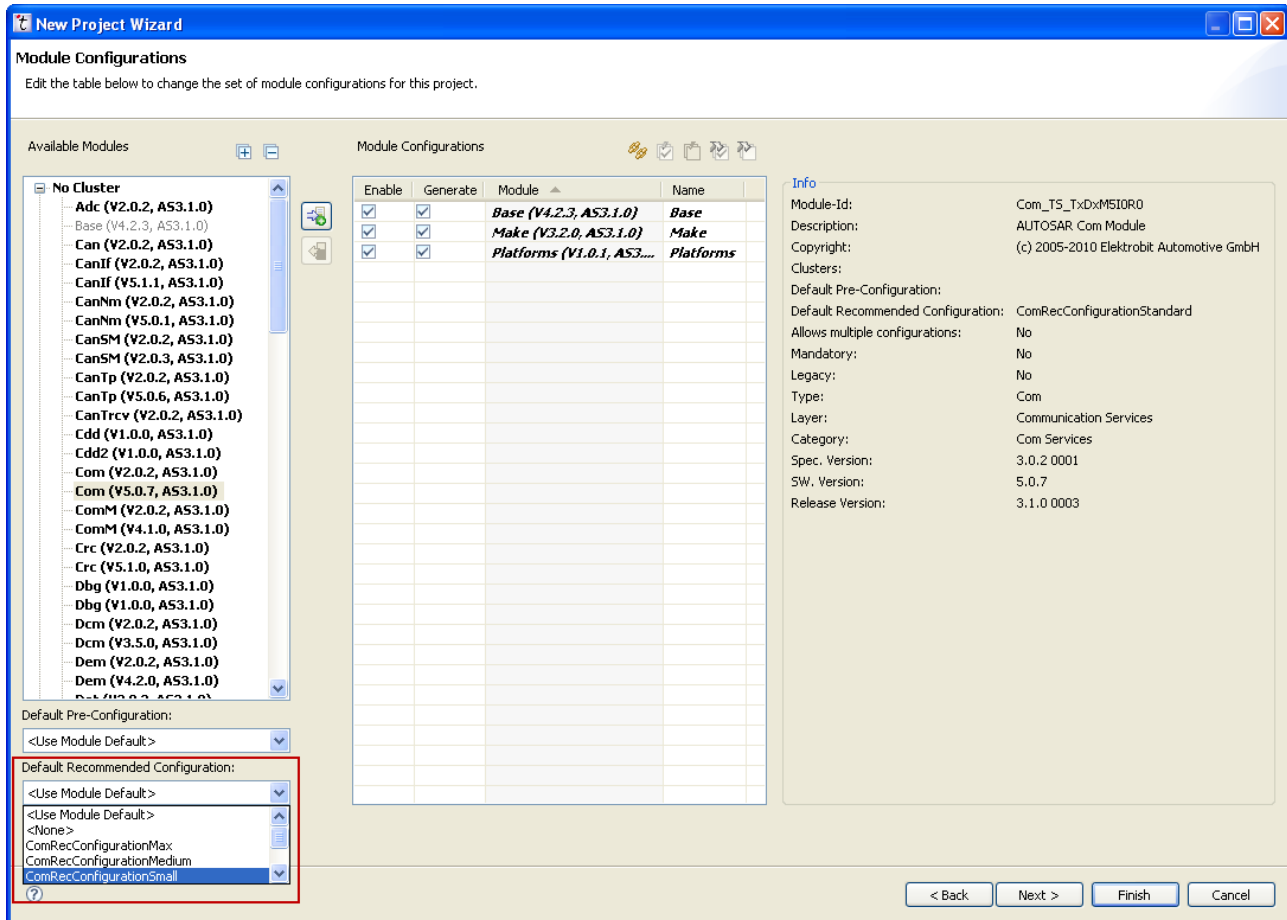


Figure 7.4. Selecting the small recommended configuration while adding the Com module to your project

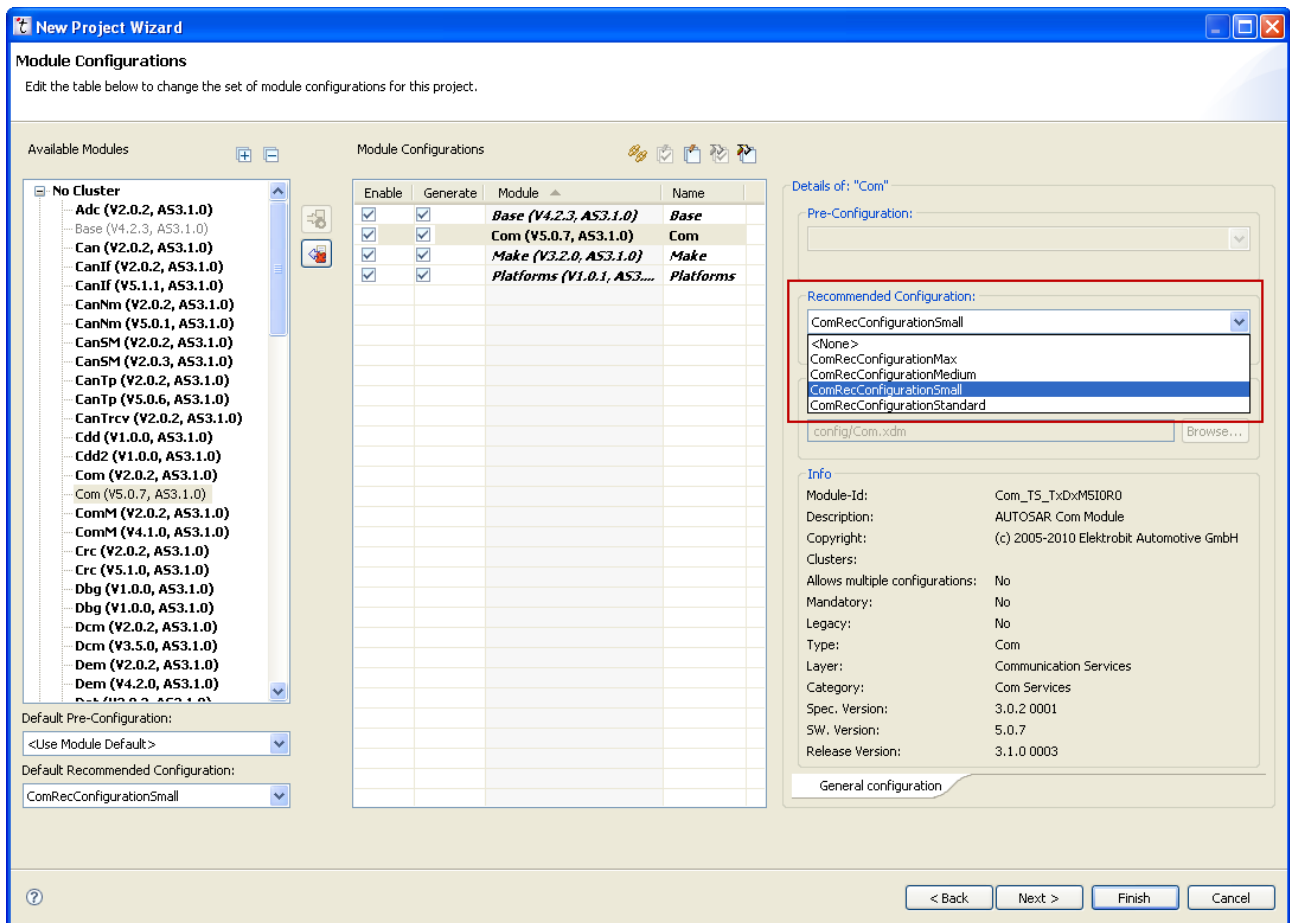


Figure 7.5. Selecting the small recommended configuration after adding the Com module to your project

7.4.4. Optimizing each parameter

When you configure a module, the EB tresos Studio element description (**Elt. Desc**) view displays a context help for each parameter.

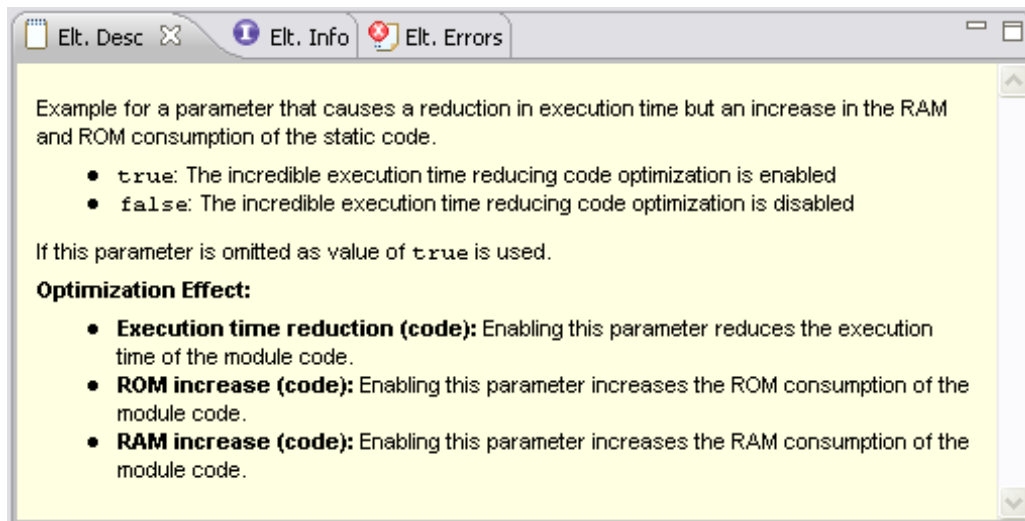


Figure 7.6. The optimization information in the **Elt. Desc** view

This information is also available in the document EB tresos AutoCore module references for example, as shown below:

Parameter Name	ComExampleExecutionTimeReductionParameter
Description	<p>Example for a parameter that causes a reduction in execution time but an increase in the RAM and ROM consumption of the static code.</p> <ul style="list-style-type: none"> ▶ <code>true</code>: The incredible execution time reducing code optimization is enabled ▶ <code>false</code>: The incredible execution time reducing code optimization is disabled <p>If this parameter is omitted as value of <code>true</code> is used.</p> <p>Optimization Effect:</p> <ul style="list-style-type: none"> ▶ Execution time reduction (code): Enabling this parameter reduces the execution time of the module code. ▶ ROM increase (code): Enabling this parameter increases the ROM consumption of the module code. ▶ RAM increase (code): Enabling this parameter increases the RAM consumption of the module code.
Multiplicity	1..1
Type	BOOLEAN
Default value	true
Configuration class	PreCompile:
Origin	Elektrobit Automotive Software

Figure 7.7. The optimization information in the document EB tresos AutoCore module references

If such a parameter provides optimization possibilities, the context help displays the following additional optimization information:

Optimization Effect:

- ▶ **Execution time reduction (code)**: Enabling this parameter reduces the execution time of the module code.
- ▶ **ROM increase (code)**: Enabling this parameter increases the ROM consumption of the module code.
- ▶ **RAM increase (code)**: Enabling this parameter increases the RAM consumption of the module code.

The following example shows a `FrSM` parameter in the **Generic Editor** view and its corresponding optimization effect. To achieve the optimization effect described in the **Elt. Desc** view, click the checkbox to the right of the parameter.

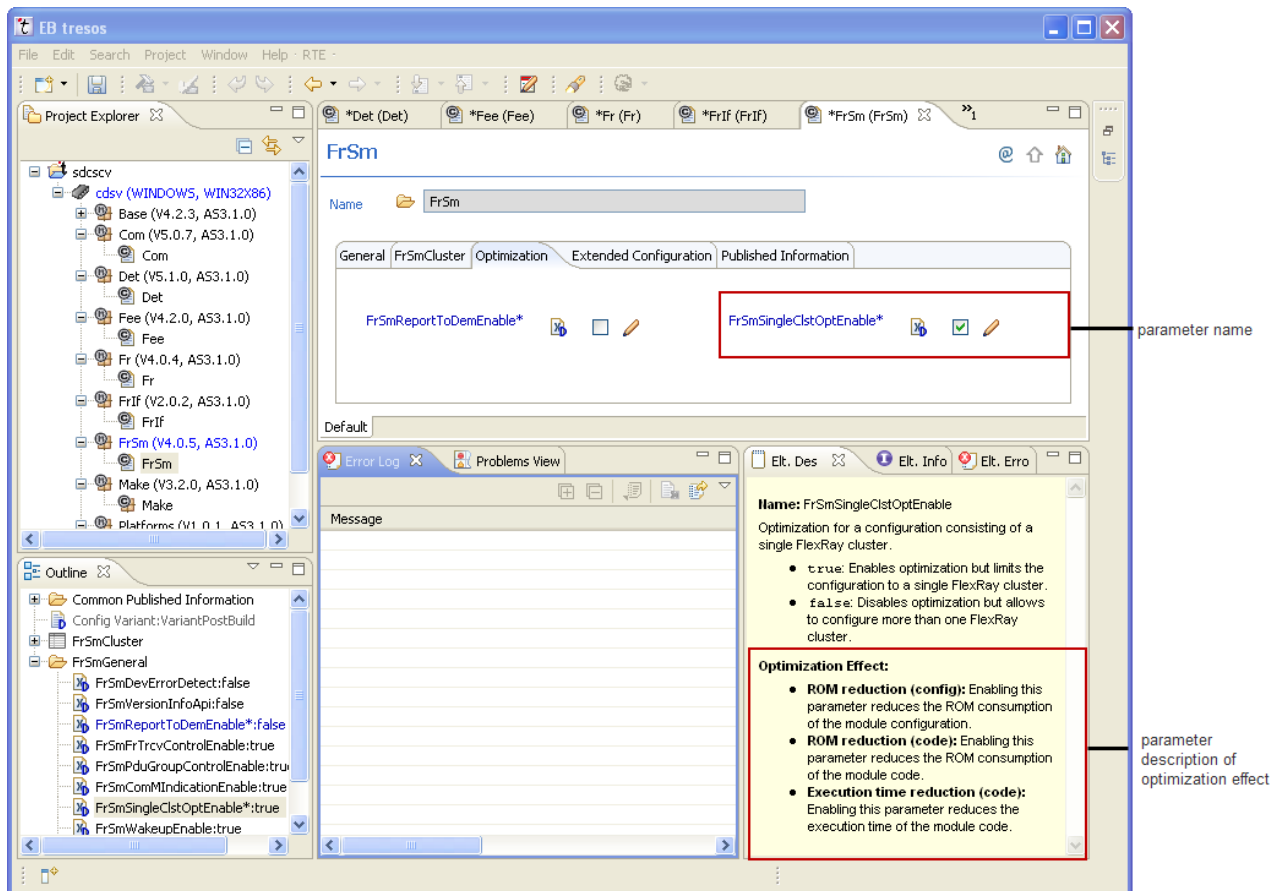


Figure 7.8. Example: optimization of a parameter and its effect on the ROM, RAM, and code execution time

7.4.5. Optimizing your project as a whole

When you have configured all EB tresos AutoCore stacks for your ECU and you are ready to generate code and test it, you can take some further optimization steps.

The following instructions guide you through EB tresos AutoCore-wide optimization issues rather than module-specific ones.

7.4.5.1. Switching off development error detection (Det) checks

Certain checks are performed at the start of each public function and are disabled if the Det checks are switched off:

- ▶ pointer checks: check pointers to be not NULL
- ▶ check the parameter ranges

- ▶ check if the function is called within a valid state

To improve the ROM code size and the runtime behavior, check whether you still need `Det` checks in any module. If you no longer need to trace development errors,

- ▶ disable the `Det` checks from all modules, and
- ▶ remove the `Det` module itself from the project.

To disable `Det` checks in your modules, ensure that the parameter labeled **Enable development Error Detection** is unchecked.

7.4.5.2. Switching off the `GetVersionInfo()` API

All AUTOSAR modules have the parameter **Enable Version Info API**. To improve the ROM size code, switch off **Enable Version Info API** in all modules.

By switching of this parameter, the function which provides the version information is no longer present in your code.

7.4.5.3. Adjusting the exclusive areas configuration

To protect critical sections of code, basic software modules use exclusive areas that must be configured in the BSW scheduler (`SchM`) of the `Rte`. See the EB tresos AutoCore Generic RTE documentation for details.

The default mapping for exclusive areas that works in all system environments but which is not optimized for performance is the mapping to **All Interrupt Blocking**.

One possibility to optimize exclusive areas is to configure the exclusive area to use in-line macros i.e. set the implementation mechanism of a basic software module's exclusive area to the value **EB Fast Lock**

Using this optimization option, the locking mechanism is still used for the basic software module but you can choose the implementation to be a small and fast in-line assembler macro. This option is only feasible if the module runs inside a trusted OS partition.

8. Bibliography

Bibliography

- [1] *Information Technology - Open Systems Interconnection - Basic Reference Model: The Basic Model*,
1, rue de Varembe, Case postale 56
1211 Geneva 20, Switzerland , Publish date: 1994, Issue Version ISO/IEC 7498-1, Publisher: ISO
(International Organization for Standardization)
- [2] *OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection*, Author: Zimmermann, Pages: 425 to 432,
- [3] *AUTOSAR Specification of ECU Configuration*, Issue Document number 087. Document version 3.1.0, Revision 0002, Part of release 4.0.3, Publisher: AUTOSAR
- [4] *Road Vehicles - Diagnostics on Controller Area Networks (CAN) - Part 2: Network Layer Services*,
1, rue de Varembe, Case postale 56
1211 Geneva 20, Switzerland , Publish date: 2003, Issue Version ISO/DIS 15765-2.2, Publisher: ISO
(International Organization for Standardization)
- [5] *Road Vehicles - Communication on FlexRay - Part 2: Communication Layer Services*,
1, rue de Varembe, Case postale 56
1211 Geneva 20, Switzerland , Publish date: 2010, Version ISO 10681-2:2010, Publisher: ISO (International Organization for Standardization)
- [6] *AUTOSAR Specification of Memory Mapping*, Issue Version 1.4.0, Release 4.0, Revision 3, Publisher: AUTOSAR
- [7] *AUTOSAR Model Persistence Rules for XML*, Issue Version 2.1.2, Release 3.0, Revision 0001, Publisher: AUTOSAR
- [8] *AUTOSAR Specification of RTE*, Issue Version 3.2.0, Release 4.0, Revision 3, Publisher: AUTOSAR
- [9] *AUTOSAR Specification of BSW Module Description Template*, Issue Version 2.2.0, Release 4.0, Revision 3, Publisher: AUTOSAR
- [10] *AUTOSAR Specification of Software Component Template*, Issue Version 4.2.0, Release 4.0, Revision 3, Publisher: AUTOSAR
- [11] *AUTOSAR Guide to BSW Distribution*, Issue Release 4.0, Revision 3, Publisher: AUTOSAR
- [12] *AUTOSAR Requirements on Operating Systems*, Issue Release 4.0, Revision 3, Publisher: AUTOSAR
- [13] *AUTOSAR Specification of Operating Systems*, Issue Release 4.0, Revision 3, Publisher: AUTOSAR



- [14] *Validity of the single processor approach to achieving large scale computing capabilities*, Issue April 18-20, 1967, Publisher: Amdahl, Gene M.
- [15] *AUTOSAR Specification of ECU State Manager*, Issue Release 4.0, Revision 3, Publisher: AUTOSAR
- [16] *AUTOSAR Specification of Basic Software Mode Manager*, Issue Release 4.0, Revision 3, Publisher: AUTOSAR