Discover the Experience

EB tresos® Studio documentation

release 2009.a

EB tresos® Studio documentation

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

Technical support

Europe

Phone: +49-9131-7701-6060

Japan

Phone: +81-3-5775-6160

USA

Phone: +1-888-346-3813

Support URL

http://automotive.elektrobit.com/support

Legal notice

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.

ProOSEK® and tresos® are registered trademarks of Elektrobit Automotive GmbH.

All trademarks used in this document are the property of their respective owners.
© 2009 Elektrobit Automotive GmbH

Table of Contents

Begin here 20
1. If you are upgrading from a previous version 20
2. If you are using EB tresos Studio for the first time 20
3. If you need help/more information 21
1. About this documentation 22
1.1. Introduction 22
1.2. Target group 22
1.1. Required knowledge and system environment 23
1.1.1. Required system environment 23
1.3. Typography and style conventions 23
1.4. Naming conventions 25
1.4.1. Autosar XML schema 25
1.4.2. Configuration parameter names 25
1.5. EB tresos Studio naming conventions 26
1.5.1. EB tresos Studio command shell 26
2. EB tresos product line support 27
3. Background information 28
3.1. Overview 28
3.2. The Autosar concept 28
3.3. EB tresos Studio: configuration and generation tool for ECU standard software 29
3.3.1. The configuration and generation tool 29
3.3.2. Usage 30
3.3.3. Benefits 31
3.3.4. Features 31
4. EB tresos Studio user's guide 34
4.1. Overview 34
4.2. Background knowledge 35
4.2.1. File and directory structure 35
4.2.2. EB tresos Studio projects and workspaces 36
4.2.2.1. The Workspace 36
4.2.2.2. Projects 37
4.2.2.3. File content types 37
4.2.2.3.1. Content types for XDM files 37
4.2.2.3.2. Autosar content types 38
4.3. Using EB tresos Studio for the first time 39
4.3.1. Starting the graphical user interface 39
4.3.2. First launch 40
4.4. The GUI main window 41
4.4.1. Overview 41

4.4.2. Menubar and Toolbar 42
4.4.2.1. File menu 44
4.4.2.2. Edit menu 45
4.4.2.3. Search menu 46
4.4.2.4. Project menu 47
4.4.2.5. Window menu 48
4.4.2.6. Help menu 48
4.4.3. Project Explorer view 49
4.4.3.1. Project and ECU configuration states 51
4.4.3.2. Layering and grouping of module configurations 51
4.4.3.3. Changing the visible layers 52
4.4.3.4. Collapsing the Project Explorer tree 57
4.4.3.5. Synchronizing the Project Explorer with the Editor view 58
4.4.3.6. Indicating different project states with colors 59
4.4.4. Configuration Editor 60
4.4.4.1. Node Outline view 61
4.4.4.2. Element Outline views 62
4.4.4.2.1. Element Errors view 62
4.4.4.2.2. Element Info view 63
4.4.4.2.3. Element Description view 64
4.4.5. Error Log view 65
4.4.5.1. Changing the display settings of the Error Log view 67
4.4.5.2. Filtering Error Log messages by their severity level 68
4.4.5.3. Restricting the amount of shown Error Log messages and the backup logfile size 69
4.4.5.4. Copying Error Log entries into text documents 70
4.4.6. Problems view 71
4.4.6.1. Filtering Problems view messages by their severity level 73
4.4.6.2. Limit the number of entries shown in the Problems View 73
4.5. Changing the EB tresos Studio setup 74
4.5.1. Managing licenses 74
4.5.1.1. Adding a license 78
4.5.1.2. Deleting a license 79
4.5.1.3. Refreshing license information 79
4.5.2. Viewing setup information 79
4.5.2.1. Finding the EB tresos Studio version, the build number, and the available features 79
4.5.2.2. Finding detailed setup information via the *Tresos Details* dialog 80
4.5.2.3. Finding the location of the installation and the workspace 82
4.5.2.4. Finding all installed modules 82
4.5.2.5. Finding all installed plug-ins 83
4.5.2.6. Finding program startup parameters and system property settings 83

4.5.2.7. Exporting a system report of your EB tresos Studio installation 84
4.5.3. Setting preferences 85
4.5.3.1. Setting general preferences 87
4.5.3.2. Setting Autosar preferences 88
4.5.3.3. Setting configuration preferences 89
4.5.3.4. Setting Error Log preferences 91
4.5.3.5. Setting License Server preferences 92
4.5.3.6. Setting colors and fonts 93
4.5.4. Working with multiple workspaces to store different projects and settings 96
4.5.4.1. Switching the workspace 96
4.5.4.2. Finding out which workspace directory is used 97
4.5.5. Customizing the main window 97
4.5.5.1. Moving views to a different location 97
4.5.5.2. Minimizing, maximizing and closing views 98
4.5.5.3. Opening a view 98
4.5.5.4. Resetting all views to the default layout 98
4.5.6. Using EB tresos Studio with version control systems 98
4.5.6.1. Placing a EB tresos Studio installation under version control systems 99
4.5.6.2. Placing a projects under version control systems 99
4.6. Working with projects 99
4.6.1. Creating a new project 99
4.6.1.1. Step 1: Starting the Project Wizard 100
4.6.1.2. Step 2: Creating a new project 102
4.6.1.3. Step 3: Configuring project properties 103
4.6.1.4. Step 4: Defining module configurations 105
4.6.1.4.1. Adding the module configurations 105
4.6.1.4.2. Editing the module configuration 106
4.6.1.4.3. Deleting module configurations from the list 106
4.6.1.4.4. Upgrading module configurations 106
4.6.1.5. Step 6: Importing data into the project (optional) 106
4.6.1.6. Step 7: Finishing set-up of the new project 107
4.6.2. Saving a project 107
4.6.3. Renaming a project 108
4.6.4. Copying a project 108
4.6.5. Deleting a project 109
4.6.6. Closing and opening projects 109
4.6.6.1. Closing projects 110
4.6.6.2. Opening a project 111
4.6.6.3. Loading a project 111
4.6.7. Viewing and changing project properties 112
4.6.7.1. Listing the supported models of the project 113
4.6.7.2. Editing the Code Generator settings 114

4.6.7.3. Editing the ECU Configuration settings 116
4.6.7.4. Viewing information about the project's modules 117
4.7. Working with ECU configurations 118
4.7.1. Viewing the list of module configurations for a project 118
4.7.2. Viewing detailed information of a module configuration 119
4.7.3. Adding new module configurations 121
4.7.4. Deleting module configurations 123
4.7.5. Disabling code generation of a module configuration 124
4.7.6. Disabling and enabling module configurations 125
4.7.6.1. Disabling module configurations 125
4.7.6.2. Enabling module configurations 126
4.8. Upgrading projects and configurations 127
4.8.1. Converting a project to a new Autosar version 127
4.8.2. Upgrading module configurations to new module versions 129
4.9. Working with the ECU configuration editor 132
4.9.1. Editing a module configuration 132
4.9.2. Using the Editor view 133
4.9.2.1. Page header, page name, and tabs 134
4.9.2.2. Sections and parameters 136
4.9.2.3. References 139
4.9.2.4. Choices 140
4.9.2.5. Tables 141
4.9.2.6. Optional elements 144
4.9.2.7. Changing multiple rows in a table 144
4.9.3. Autoconfigure wizards 145
4.9.3.1. Configuring Autoconfigure wizards 146
4.9.3.2. Running Autoconfigure wizards 147
4.9.3.2.1. Running a single autoconfigure wizard 147
4.9.3.2.2. Running multiple autoconfigure wizards 149
4.9.3.2.3. Result feedback 150
4.9.3.3. Configuring the *Calculate Default Values* wizard 151
4.9.3.4. Configuring the *Calculate Handle-Ids* wizard 152
4.9.4. Searching for configuration parameter names and their values 153
4.9.4.1. Searching in the data model 154
4.9.4.2. Displaying search results 156
4.10. Generating code for projects 158
4.10.1. Verifying a project 158
4.10.2. Generating a project 159
4.10.3. Building a project 160
4.11. Importing and exporting 161
4.11.1. Exchanging a complete EB tresos Studio project with other installations 161
4.11.1.1. Importing a project 161

4.11.1.2. Exporting a project ... 166
4.11.1.2.1. Exporting a project as copy ... 167
4.11.1.2.2. Exporting a project as archive ... 169
4.11.2. Importing and exporting configuration data ... 170
4.11.2.1. Step-by-step guide ... 170
4.11.2.1.1. Step 1: Using the Create, manage and run im- and exporters window ... 170
4.11.2.1.2. Step 2: Creating an export or import configuration ... 173
4.11.2.1.3. Step 3: Saving changes to an import or export configuration ... 174
4.11.2.1.4. Step 4: Importing or exporting project configuration data ... 175
4.11.2.2. Importing and exporting configuration data from and to the Autosar format ... 177
4.11.2.2.1. How the import works ... 177
4.11.2.2.2. Importing and exporting configuration data ... 177
4.11.2.3. Importing Autosar system descriptions ... 179
4.11.2.3.1. Overview ... 179
4.11.2.3.2. Background information ... 180
4.11.2.3.3. Importing system information ... 180
4.11.2.3.4. References ... 186
4.11.2.4. Importing from the OIL format ... 197
4.11.2.4.1. Overview ... 197
4.11.2.4.2. OIL Importer and file format background knowledge ... 197
4.11.2.4.3. Importing OIL configurations ... 198
4.11.2.4.4. Importing OIL files via the command line ... 200
4.11.2.5. Importing CAN configurations from the DBC format ... 201
4.11.2.5.1. Overview ... 201
4.11.2.5.2. Background information ... 201
4.11.2.5.3. Importing DBC configurations ... 202
4.11.2.5.4. References ... 207
4.11.2.6. Importing from the LDF format ... 227
4.11.2.6.1. Overview ... 227
4.11.2.6.2. LDF Importer and file format background knowledge ... 227
4.11.2.6.3. Importing LDF configurations ... 227
4.11.2.6.4. References ... 232
4.11.2.7. Importing from the Fibex format ... 242
4.11.2.7.1. Overview ... 242
4.11.2.7.2. Background information ... 242
4.11.2.7.3. Importing Fibex configurations ... 243
4.11.2.7.4. Retrieving PDUs from Fibex 2.x and earlier files ... 246
4.11.2.7.5. References ... 247
4.11.2.8. Importing configurations from the TDB format ... 266
4.11.2.8.1. Overview ... 266
4.11.2.8.2. Background information ... 266

4.11.2.8.3. Importing TDB configurations 266
4.11.2.8.4. TDB to Autosar parameter mapping 271
4.11.3. Background information on importing network configuration formats 271
4.11.3.1. Naming rules 271
4.11.3.1.1. Name mangling algorithm for DBC and LDF Importers 271
4.11.3.1.2. Name prefixing 272
4.11.3.1.3. Instance handling 272
4.11.3.1.4. Signal offsets 274
4.12. Working with the command line 274
4.12.1. General 274
4.12.1.1. Starting the command line 274
4.12.1.2. The command line syntax 275
4.12.1.3. System properties common to all commands 275
4.12.1.4. Changing the workspace on the command line 276
4.12.2. Commands for project-based code generation and configuration 277
4.12.2.1. Verifying a project 277
4.12.2.1.1. Syntax 277
4.12.2.1.2. System properties 277
4.12.2.1.3. Parameters 277
4.12.2.1.4. Example 278
4.12.2.2. Generating code for a project 278
4.12.2.2.1. Syntax 278
4.12.2.2.2. System properties 278
4.12.2.2.3. Options 278
4.12.2.2.4. Parameters 279
4.12.2.2.5. Example 279
4.12.2.3. Listing custom-built generation modes 279
4.12.2.3.1. Syntax 280
4.12.2.3.2. System properties 280
4.12.2.3.3. Parameters 280
4.12.2.3.4. Example 280
4.12.2.4. Executing custom-built generation modes for projects 281
4.12.2.4.1. Syntax 281
4.12.2.4.2. System properties 281
4.12.2.4.3. Parameters 281
4.12.2.4.4. Example 282
4.12.2.5. Executing autoconfigure wizards on projects 282
4.12.2.5.1. Syntax 282
4.12.2.5.2. System properties 282
4.12.2.5.3. Parameters 283
4.12.2.5.4. Example 283
4.12.2.5.5. Running the Calculate Default Values wizard on the command line 283

4.12.2.5.6. Running the Calculate Handle-Ids wizard on the command line 283
4.12.3. Commands for legacy code generation 284
4.12.3.1. Naming conventions for modules usable in legacy-mode 284
4.12.3.2. Listing all modules available for legacy-mode 284
4.12.3.2.1. Syntax 284
4.12.3.2.2. System properties 285
4.12.3.2.3. Example 285
4.12.3.3. Generating code for legacy modules 285
4.12.3.3.1. Syntax 285
4.12.3.3.2. System properties 285
4.12.3.3.3. Options 286
4.12.3.3.4. Parameters 287
4.12.3.3.5. Example 287
4.12.3.4. Verifying configurations for legacy modules 288
4.12.3.4.1. Syntax 288
4.12.3.4.2. System properties 288
4.12.3.4.3. Options 288
4.12.3.4.4. Parameters 289
4.12.3.4.5. Example 290
4.12.3.5. Listing all custom-built generation modes for legacy modules 290
4.12.3.5.1. Syntax 290
4.12.3.5.2. System properties 290
4.12.3.5.3. Example 290
4.12.3.6. Running custom-built generation modes for legacy modules 291
4.12.3.6.1. Syntax 291
4.12.3.6.2. System properties 291
4.12.3.6.3. Options 291
4.12.3.6.4. Parameters 292
4.12.3.6.5. Example 293
4.12.4. Commands for module development 293
4.12.4.1. Listing a data model 293
4.12.4.1.1. Syntax 293
4.12.4.1.2. System properties 294
4.12.4.1.3. Options 294
4.12.4.1.4. Parameters 294
4.12.4.1.5. Example 295
4.12.4.2. Creating vendor-specific module definitions (VSMD) 295
4.12.4.2.1. Syntax 295
4.12.4.2.2. System properties 296
4.12.4.2.3. Options 297
4.12.4.2.4. Parameters 298
4.12.4.2.5. Example 299

4.12.4.3. Checking vendor-specific against standard module definitions 299
4.12.4.3.1. Syntax 299
4.12.4.3.2. System properties 299
4.12.4.3.3. Parameters 300
4.12.4.3.4. Example 300
4.12.4.4. Listing all rules used for VSMD checking 301
4.12.4.4.1. Syntax 301
4.12.4.4.2. System properties 301
4.12.4.4.3. Example 301
4.12.4.5. Encrypting and singing modules 301
4.12.5. Commands for converting and merging of files 301
4.12.5.1. Merging files and converting between the XDM and Autosar formats 301
4.12.5.1.1. Syntax 302
4.12.5.1.2. System properties 302
4.12.5.1.3. Options 304
4.12.5.1.4. Parameters 305
4.12.5.1.5. Example 305
4.12.5.2. Using XDM files in different EB tresos Studio versions 305
4.12.5.2.1. Example 306
4.12.5.3. Upgrading the Autosar compliance of XDM files 306
4.12.5.3.1. Syntax 306
4.12.5.3.2. System properties 307
4.12.5.3.3. Parameters 307
5. EB tresos Studio developer's guide 308
5.1. Introduction 308
5.2. Interpretation of version information 308
5.3. Setting up a new module 308
5.3.1. Prerequisites 308
5.3.2. Creating a vendor specific module definition 309
5.3.3. Converting files 309
5.3.4. Templates for new modules 310
5.3.5. Setup of the file structure 310
5.3.6. Adapting MANIFEST.MF 310
5.3.7. Adapting plugin.xml 311
5.3.7.1. dreisoft.tresos.launcher2.plugin.module 311
5.3.7.2. dreisoft.tresos.launcher2.plugin.configuration 313
5.3.7.2.1. Parameter definition, pre-, recommended configuration 313
5.3.7.2.2. Module configuration 315
5.3.7.2.3. Registering editors 315
5.3.7.2.4. Code sample 316
5.3.7.3. dreisoft.tresos.launcher2.plugin.generator 317
5.3.7.4. Module versions 318

5.3.7.4.1. Software version ... 319
5.3.7.4.2. Specification version ... 320
5.3.7.4.3. Release version ... 320
5.3.8. Code templates ... 320
5.3.9. Adding the module to EB tresos Studio ... 320
5.4. Upgrading an Autosar 2.1 Module to 3.0 ... 321
5.4.1. Choices ... 322
5.4.2. Configuration variant ... 322
5.4.3. ADMIN-DATA ... 323
5.5. Module Transformer ... 324
5.5.1. Registering a Module Transformer ... 324
5.5.1.1. Examples ... 325
5.5.2. Implementing a Module Transformer Class ... 326
5.5.3. Documentation ... 327
5.5.4. Chaining of Module Transformers ... 327
5.6. Signing modules with the crypto command ... 327
5.6.1. Background ... 327
5.6.2. Prerequisites ... 327
5.6.3. Crypto command ... 328
5.7. The DataModel ... 328
5.7.1. Schematic-tree ... 331
5.7.1.1. Example ... 331
5.7.1.2. Features of Schema-Nodes ... 332
5.7.2. Data-Tree ... 333
5.7.3. Node-Attributes ... 333
5.7.4. List-Representation within the DataModel ... 334
5.7.5. Choice-representation within the DataModel ... 335
5.8. The XDM format ... 335
5.8.1. Concepts ... 336
5.8.2. Top level structure ... 336
5.8.3. Nodes ... 337
5.8.4. Attributes ... 338
5.8.4.1. Passive attributes ... 338
5.8.4.2. Generic attributes ... 339
5.8.4.2.1. AdminData ... 339
5.8.4.2.2. Implementation Config Class ... 340
5.8.4.3. Automatic attributes ... 340
5.8.4.3.1. Bound attribute ... 341
5.8.4.3.2. Match attribute ... 342
5.8.4.3.3. Range attribute ... 343
5.8.4.3.4. XPath attribute ... 344
5.8.4.3.5. XDM attribute examples ... 344

5.8.4.3.6. Restrictions 350
5.8.4.4. Data attributes 350
5.8.4.5. Common attributes 351
5.8.5. Generic nodes 351
5.8.5.1. link 351
5.8.5.1.1. Attributes 351
5.8.5.1.2. Valid children 352
5.8.6. Schema-nodes 352
5.8.6.1. ctr 352
5.8.6.1.1. Attributes and data attributes 353
5.8.6.1.2. Valid children 353
5.8.6.2. chc 353
5.8.6.2.1. Attributes and data attributes 354
5.8.6.2.2. Valid children 354
5.8.6.3. lst 354
5.8.6.3.1. Attributes and data attributes 354
5.8.6.3.2. Valid children 354
5.8.6.4. var 355
5.8.6.4.1. Attributes and data attributes 356
5.8.6.4.2. Valid children 356
5.8.6.5. ref 356
5.8.6.5.1. Attributes and data attributes 357
5.8.6.5.2. Valid children 357
5.8.7. Data-nodes 357
5.8.7.1. ctr 358
5.8.7.1.1. Attributes 358
5.8.7.1.2. Valid Children 358
5.8.7.2. chc 358
5.8.7.2.1. Attributes 358
5.8.7.2.2. Valid Children 358
5.8.7.3. lst 358
5.8.7.3.1. Attributes 359
5.8.7.3.2. Valid Children 359
5.8.7.4. var 359
5.8.7.4.1. Attributes 359
5.8.7.4.2. Valid Children 359
5.8.7.5. ref 359
5.8.7.5.1. Attributes 360
5.8.7.5.2. Valid Children 360
5.8.8. Factories 360
5.8.9. Path addressing 360
5.9. Mapping Autosar ECU Configuration to XDM 362

5.9.1. Node mapping ... 362
5.9.1.1. Schema-Nodes ... 362
5.9.1.1.1. AR-PACKAGE ... 362
5.9.1.1.2. MODULE-DEF ... 362
5.9.1.1.3. PARAM-CONF-CONTAINER-DEF ... 363
5.9.1.1.4. MULTIPLE-CONFIGURATION-CONTAINER ... 363
5.9.1.1.5. BOOLEAN-PARAM-DEF ... 363
5.9.1.1.6. INTEGER-PARAM-DEF ... 364
5.9.1.1.7. FLOAT-PARAM-DEF ... 364
5.9.1.1.8. STRING-PARAM-DEF ... 364
5.9.1.1.9. LINKER-SYMBOL-DEF ... 365
5.9.1.1.10. FUNCTION-NAME-DEF ... 365
5.9.1.1.11. ENUMERATION-PARAM-DEF ... 365
5.9.1.2. CHOICE-CONTAINER-DEF ... 366
5.9.1.3. CHOICE-REFERENCE-DEF ... 366
5.9.1.4. REFERENCE-DEF ... 367
5.9.1.5. SYMBOLIC-NAME-REFERENCE-DEF ... 367
5.9.1.6. INSTANCE-REFERENCE-DEF ... 368
5.9.1.7. FOREIGN-REFERENCE-DEF ... 368
5.9.1.8. DERIVED-BOOLEAN-PARAM-DEF ... 369
5.9.1.9. DERIVED-INTEGER-PARAM-DEF ... 369
5.9.1.10. DERIVED-FLOAT-PARAM-DEF ... 369
5.9.1.11. DERIVED-STRING-PARAM-DEF ... 370
5.9.1.12. DERIVED-ENUMERATION-PARAM-DEF ... 370
5.9.1.13. Data-Nodes ... 370
5.9.1.13.1. ECU-CONFIGURATION ... 370
5.9.1.13.2. MODULE-CONFIGURATION ... 371
5.9.1.13.3. CONTAINER ... 371
5.9.1.13.4. Choices ... 371
5.9.1.13.5. BOOLEAN-VALUE ... 372
5.9.1.13.6. INTEGER-VALUE ... 373
5.9.1.13.7. FLOAT-VALUE ... 373
5.9.1.13.8. STRING-VALUE ... 374
5.9.1.13.9. LINKER-SYMBOL-VALUE ... 374
5.9.1.13.10. FUNCTION-NAME-VALUE ... 374
5.9.1.13.11. ENUMERATION-VALUE ... 375
5.9.1.13.12. REFERENCE-VALUE ... 375
5.9.1.13.13. INSTANCE-REFERENCE-VALUE ... 376
5.9.1.13.14. Example: ... 377
5.9.2. Attribute Mapping ... 377
5.9.2.1. CONFIGURATION-CLASS-AFFECTION ... 378
5.9.3. XPath-addressing of ECU-CD using ECU-PD ... 378

5.9.3.1. Path in Template 380
5.9.4. Top level Structure 381
5.9.5. List-Nodes 383
5.9.5.1. Optional elements 384
5.9.5.2. Optional elements within the commandline 385
5.9.5.3. Optional elements within the GUI 385
5.9.6. Choice-Nodes 386
5.9.7. References 387
5.10. XPath within the DataModel 387
5.10.1. XPath Console 388
5.10.2. Conventions 389
5.10.3. Data-Types within XPath 389
5.10.4. Additional Data-Types for the DataModel 390
5.10.5. DataModel-Nodes and their XPath-Value 390
5.10.6. Example-Tree 391
5.10.7. XPath expressions 391
5.10.8. XPath predicates 392
5.10.9. XPath operations 393
5.10.10. XPath functions 394
5.10.10.1. Node-Set functions 395
5.10.10.2. String-functions 396
5.10.10.3. Boolean-functions 398
5.10.10.4. Number-functions 399
5.10.10.5. Additional functions provided by the XPath Engine 399
5.10.10.5.1. Node functions 400
5.10.10.5.2. Text functions 402
5.10.10.5.3. Number functions 407
5.10.10.5.4. Bit functions 411
5.10.10.5.5. Variable functions 413
5.10.10.5.6. License functions 413
5.10.10.6. Additional XPath-Functions for Autosar 413
5.10.11. Pitfalls when using XPath 415
5.10.11.1. Access to non-existing nodes in comparisons 416
5.10.11.2. Using node values and variables 416
5.10.11.3. Values that represent a logical false in XPath 417
5.11. GUI Annotation in XDM 417
5.11.1. Generic Configuration Editor Layout 417
5.11.2. Module GUI configuration 419
5.11.3. GUI Elements 420
5.11.3.1. Attributes 420
5.11.3.2. Boolean values 421

5.11.3.3. Integer Values, Float Values, String Values, Enumerations and References ... 426
5.11.3.4. Container ... 429
5.11.3.5. Choices ... 434
5.11.3.6. Lists ... 440
5.12. The Code Generator ... 446
5.12.1. General concepts ... 446
5.12.1.1. Addressing nodes using XPath ... 447
5.12.1.2. Identical generated files ... 447
5.12.2. Codetemplate Console ... 447
5.12.3. Comments ... 448
5.12.4. Evaluating variables and expressions ... 449
5.12.5. Including other files ... 449
5.12.6. Conditions ... 450
5.12.7. Loops and selects ... 450
5.12.8. Leaving loops ... 451
5.12.9. Variables and macros ... 452
5.12.10. Spaces, newlines and indentation ... 455
5.12.10.1. Tab stops ... 455
5.12.10.2. Whitespaces ... 455
5.12.10.3. Newlines ... 455
5.12.10.4. Auto-spacing ... 455
5.12.10.5. Indentation ... 456
5.12.11. Excluding files from the generation ... 457
5.12.12. Asserts, errors and warnings ... 457
5.13. The EPC file generator ... 459
5.14. Using external generators ... 460
5.14.1. Features ... 460
5.14.2. Functionality ... 461
5.14.3. Prerequisites ... 461
5.14.4. Adding the Generator to the plug-in ... 461
5.14.5. Registering an External Generator ... 462
5.14.6. Generator Commandline ... 464
5.14.7. Parsing the Output of the Generator ... 466
5.15. ECU Resource Manager ... 469
5.15.1. Registering ECU Resource Properties ... 469
5.15.1.1. Advanced properties ... 471
5.15.2. ECU Resource Finder ... 471
5.15.2.1. Advanced properties ... 472
5.15.2.2. Default ECU Resource Finder Implementation ... 472
5.15.3. ECU Resource XPath Functions ... 475
5.16. Creating a module with Eclipse ... 475

5.16.1. Installation and configuration ... 475
5.16.2. Extension points and their description ... 480
5.16.3. Errors and warnings within the PDE ... 484
5.16.4. Creating a EB tresos Studio ModulePlugIn ... 484
5.16.4.1. Schema-file and templates ... 484
5.16.4.2. Extension points ... 485
5.16.4.2.1. Extension point dreisoft.tresos.launcher2.plugin.module ... 485
5.16.4.2.2. Extension point dreisoft.tresos.launcher2.plugin.configuration ... 486
5.16.4.2.3. Extension point extension point dreisoft.tresos.launcher2.plugin.module ... 486
5.16.5. Starting EB tresos Studio from Eclipse ... 487
5.17. The NG Generator Framework ... 489
5.17.1. Generation Steps ... 490
5.17.2. Module-Independent Generators ... 490
5.17.3. JET Support ... 491
5.17.4. Standalone JET compiler ... 492
5.17.5. C Data Structures Generator (CDS) ... 492
5.17.6. Apache Ant Support ... 493
5.17.7. Source Delivery ... 498
5.18. Setting Up Eclipse ... 498
5.18.1. Installation And Version ... 498
5.18.2. Installation of EB tresos Studio Eclipse Tooling ... 499
5.18.3. Setting Up The Target Platform ... 500
5.18.4. Setting Up A Run Configuration ... 501
5.19. National Language Support (NLS) / Operation Status (OS) ... 503
5.19.1. National Language Support (NLS) ... 503
5.19.1.1. Workflow for NLS-support in Java Code ... 504
5.19.1.2. Config File Format ... 505
5.19.1.3. NLS Examples ... 508
5.19.1.3.1. Example for the usage in a Plugin.xml ... 508
5.19.1.3.2. NLS source code usage example ... 509
5.19.2. APIOperationStatus ... 510
5.19.2.1. Glossary ... 511
5.19.2.2. Functionality ... 511
5.19.2.3. Workflow for OS-support in Java Code ... 512
5.19.2.4. Config File Format ... 513
5.19.2.5. Operation Status Examples ... 516
5.19.3. Calling The NLS/OS-Tool ... 517
5.20. Public Api Use-Cases ... 518
5.20.1. General Preparations To Use The Public API ... 518
5.20.2. Using The Public API Generator ... 518
5.20.2.1. Registering A Public API Generator ... 519

5.20.2.2. Implementing A Public API Generator ... 519
5.20.3. Using Custom XPath Functions ... 520
5.20.3.1. Registering Custom XPath Functions ... 521
5.20.3.2. Implementing Custom XPath Functions ... 521
5.21. Guided configuration API ... 523
5.21.1. Purpose ... 523
5.21.2. Knowledge required ... 523
5.21.3. Plug-ins required ... 523
5.21.4. Design overview ... 524
5.21.4.1. Wizard type overview ... 527
5.21.4.1.1. Stand-alone editor ... 529
5.21.4.1.2. Dialog ... 530
5.21.4.1.3. Autoconfiguration wizard ... 531
5.21.4.1.4. Custom module editor ... 533
5.21.4.2. Selection Context ... 534
5.21.4.2.1. ECUConfigContext ... 534
5.21.4.2.2. ModuleConfigContext ... 535
5.21.4.2.3. SystemConfigContext ... 535
5.21.4.3. Lifecycle ... 536
5.21.4.4. Data representation ... 537
5.21.4.4.1. The MementoOperationHandler ... 538
5.21.4.4.2. The Memento Interface ... 539
5.21.4.4.3. MementoFactory ... 539
5.21.4.5. GUI abstraction ... 539
5.21.4.5.1. Multi-page support ... 540
5.21.4.5.2. Layouting ... 540
5.21.4.5.3. Validating ... 540
5.21.4.5.4. Navigating ... 540
5.21.4.5.5. Predefined widgets ... 541
5.21.4.5.6. Widget lifecycle ... 545
5.21.4.5.7. ResultWidget ... 546
5.21.4.6. PushService ... 547
5.21.4.7. Locking mechanism ... 547
5.21.5. API ... 548
5.21.6. Demos ... 549
5.21.6.1. Setting up the demo plug-in ... 549
5.21.6.1.1. Making the Sidebar view visible ... 550
5.21.6.1.2. Customizing the view ... 550
5.21.6.1.3. Setting up the data ... 551
5.21.6.2. Demo1 ... 552
5.21.6.2.1. Overview ... 552
5.21.6.2.2. Restricting the visibility ... 552

5.21.6.2.3. Using multi-page support ... 553
5.21.6.2.4. Using XForm GUI description ... 553
5.21.6.2.5. Validating ... 553
5.21.6.2.6. Navigating ... 553
5.21.6.2.7. Using a custom MementoFactory ... 554
5.21.6.3. Demo2 ... 554
5.21.6.3.1. Overview ... 554
5.21.6.3.2. Restricting the visibility ... 555
5.21.6.3.3. Creating a custom widget ... 555
5.21.6.3.4. Using the PushService ... 556
5.21.6.3.5. Updating widget enablement ... 556
5.21.6.4. Demo3 ... 556
5.21.6.4.1. Overview ... 556
5.21.6.4.2. Using the PushService ... 558
5.21.6.4.3. Registering a custom ResultWidget ... 558
5.21.6.5. Demo4 ... 559
5.21.6.5.1. Overview ... 559
5.21.6.5.2. Restricting the visibility ... 560
5.21.6.5.3. Displaying predefined widgets ... 560
5.21.6.5.4. Validating ... 560
5.21.6.5.5. Using selection events ... 560
5.21.6.6. Demo5 ... 562
5.21.6.6.1. Overview ... 562
5.21.6.6.2. Using multi-page support ... 563
5.21.6.6.3. Validating ... 564
5.21.6.7. Demo6 ... 564
5.21.6.7.1. Overview ... 564
5.21.6.7.2. Restricting the visibility ... 564
5.21.6.7.3. Using hierarchical Sidebar labels ... 565
5.21.7. Common use cases ... 567
5.21.7.1. Registering a wizard ... 567
5.21.7.2. Registering the docking point ... 569
5.21.7.2.1. Sidebar editor and dialog ... 569
5.21.7.2.2. Custom module editor ... 571
5.21.7.2.3. Autoconfiguration wizard ... 571
5.21.7.3. Restricting the visibility of a wizard ... 573
5.21.7.3.1. Via extension point ... 573
5.21.7.3.2. Via Backend implementation ... 573
5.21.7.4. Using XForms as GUI description ... 574
5.21.7.5. Creating the page ... 575
5.21.7.6. Implementing a custom widget ... 576
5.21.7.7. Validating the data ... 577

5.21.7.8. Using selection events 578
5.21.7.9. Creating a MementoFactory 579
5.21.7.10. Using the PushService 579
5.21.7.10.1. Registering a PushOperation 579
5.21.7.10.2. Sending a PushEvent 580
5.21.7.11. Registering a custom ResultWidget 581
5.22. Installing the Public Api Demo Plugin 581
6. Bibliography 582
Glossary 583
Index 608

Begin here

1. If you are upgrading from a previous version

The EB tresos Studio release notes are not part of this documentation but located in a separate pdf inside the same folders as this documentation.

Please refer to the EB tresos Studio release notes to find out about:

- Version information
- Open issues and deviations
- Changes from one version to another

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

If you are a first time user of EB tresos Studio, you may want to get familiar with some of the concepts behind Autosar at Section 3.2: The Autosar concept

- What is EB tresos Studio?

  The best way to find out more about EB tresos Studio is to read through Section 3.3: EB tresos Studio: configuration and generation tool for ECU standard software to familiarize yourself with EB tresos Studio.

- How do I use the tool?

  The EB tresos Studio user's guide is the place to start if you want to:

  - Get to know the EB tresos Studio GUI (Section 4.4: The GUI main window).
  - Use the EB tresos Studio GUI to edit an ECU configuration (Section 4.9: Working with the ECU configuration editor).
  - Use the importing functions to import DBC, LDF, or FIBEX files (Section 4.11: Importing and exporting).
  - Use the EB tresos Studio tool on the command line (Section 4.12: Working with the command line).

- I want to develop my own Autosar plug-ins - can I do this with EB tresos Studio?

  Yes, you can. The EB tresos Studio developer's guide is the place to start for you (Chapter 5: EB tresos Studio developer's guide). From it you will learn how to develop your own plug-in(s). The EB tresos Studio developer's guide provides instructions, background knowledge, and parameter references to enable you to add your own module(s) to the Autosar core.

3. If you need help/more information

- Chapter 2: EB tresos product line support

  Receive technical support via e-mail or phone from the Chapter 2: EB tresos product line support hotline.

- Chapter 1: 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.

- Index

  You cannot find what you are looking for? Try the Index list (alphabetically sorted).

- Glossary

  You do not understand what a word or abbreviations means? Find out in the Glossary.

- Chapter 6: Bibliography

  Would you like to read more detailed information? Find bibliographic references and further reading suggestions in the Chapter 6: Bibliography.

1 About this documentation

1.1. Introduction

Welcome to the EB tresos Studio documentation.

This chapter `About this documentation` provides you with

- 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.

Note

Please note that there is a separate documentation for the EB tresos AutoCore, if you also purchased and installed the EB tresos AutoCore modules.

1.2. Target group

EB tresos AutoCore and EB tresos Studio are the right tools for you, if you:

- need a platform for your Autosar-compliant software components
- want to configure your Autosar basic software
- want to develop your own modules within an Autosar stack

1.1. Required knowledge and system environment

1.1.1. Required system environment

Item	Requirement
operating system	Microsoft Windows 2000/XP
processor	Intel Pentium or compatible, 1 GHz
memory	500 MB
drives	CD-Rom drive for the installation CD
connectors	USB port, if dongled licenses are used
network	Network connection, if network licenses are used

Table 1.1. Minimum system requirements

1.3. Typography and style conventions

Throughout the documentation you will see that words and phrases are displayed in bold or italic fint, or in Monospace font. To find out what these conventions mean, please consult the following table. All default text is written in Arial Regular font without any markup.

Convention	Item is used	Example
Arial italics	to define new terms	The *basic building blocks* of a configuration are module configurations.
Arial italics	to emphasize	If your project's release version is mixed, all content types are available. It is thus called *mixed version*.
Arial italics	to indicate that a term is explained in the glossary	...exchanges *protocol data unit*s (*PDU*s) with its peer instance of other ECUs.
Arial boldface	for menus and submenus	Choose the **Options** menu.
Arial boldface	for buttons	Select **OK**.
Arial boldface	for keyboard keys	Press the **Enter** key
Arial boldface	for keyboard combination of keys	Press **Ctrl**-**Alt**-**Delete**
Arial boldface	for commands	Convert the XDM file to the newer version by using the **legacy convert** command.

Convention	Item is used	Example
Monospace font (Courier)	for file and folder names, also for chapter names	Put your script in the `function_name\abcfolder`
Monospace font (Courier)	for code	`CC_FILES_TO_BUILD =(PROJECT_PATH)\source\network\can_node.c` `CC_FILES_TO_BUILD += $(PROJECT_PATH)\source\network\can_config.c`
Monospace font (Courier)	for function names, methods, or routines	The cos function finds the cosine of each array element. Syntax line example is `MLGetVar ML_var_name`
Monospace font (Courier)	for user input/indicates variable text	Enter a `three-digit prefix` in the menu line.
Square brackets [ ]	denote optional parameters; for command syntax with optional parameters	`insertBefore [<opt>]`
Curly brackets {}	denote mandatory parameters; for command syntax with mandatory parameters (in curly brackets)	`insertBefore {<file>}`
Three dots …	indicate further parameters; for command syntax, indicates further parameters	`insertBefore [<opt>…]`
A vertical bar |	to separate parameters in a list from which one parameters must be chosen or used; for command syntax, indicates a choice of parameters	`allowinvalidmarkup {on|off}`
Warning	to show information vital for the success of your configuration	
Note	to give additional important information on the subject	
Tip	to provide helpful hints and tips	

1.4. 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.

1.4.1. 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 [11]:

- 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.

1.4.2. 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`.

1.5. EB tresos Studio naming conventions

1.5.1. EB tresos Studio command shell

System commands and properties in the EB tresos Studio command shell are spelled with mixed lower- and upper-case letters, e.g. -DmergeConfigs.

Where the Autosar XML schema uses hyphens (–) to separate parameter parts, EB tresos Studio uses the camelback case to separate different parts of the parameters.

Define the `SHORT-NAME` of the parameter `ECU-PARAMETER`. Example: `-DecuParamDefName=<name>`

Example 1.1. Example instructions that use the Autosar XML schema and the EB tresos Studio spelling convention

2 EB tresos product line support

Europe
Phone: +49-9131-7701-6060

Japan
Phone: +81-3-5775-6160

USA
Phone: +1-888-346-3813

Support URL
http://automotive.elektrobit.com/support

3 Background information

3.1. Overview

The chapter `Background information` is for first-time users of EB tresos Studio 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 Studio 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 andvanced users of EB tresos Studio to figure out why some things are the way they are, they are primarily aimed at newcomers to Autosar and EB tresos Studio and those, not very familiar with the graphical user interface.

3.2. The Autosar concept

Autosar (*Aut*omotive *O*pen *S*ystem *Ar*chitecture) 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.

3.3. EB tresos Studio: configuration and generation tool for ECU standard software

3.3.1. The configuration and generation tool

Since Autosar relies on generating code according to user requirements, a configuration tool such as EB tresos Studio is needed to configure standard software components and to generate specific ECUs.

The tool follows the Autosar methodology, but can be used for much more than just basic software modules of this standard.

The open concept of its code generator engines, parameter verification language, data model access and GUI allows you to seamlessly integrate EB tresos Studio into existing tool-chains and workflows.

It helps you to deploy standard software components within your organization in an efficient and easy-to-use manner.

The main concepts of EB tresos Studio are to follow the Autosar methodology, provide open and documented interfaces, help to avoid user-errors and create an easy-to-use tool environment.

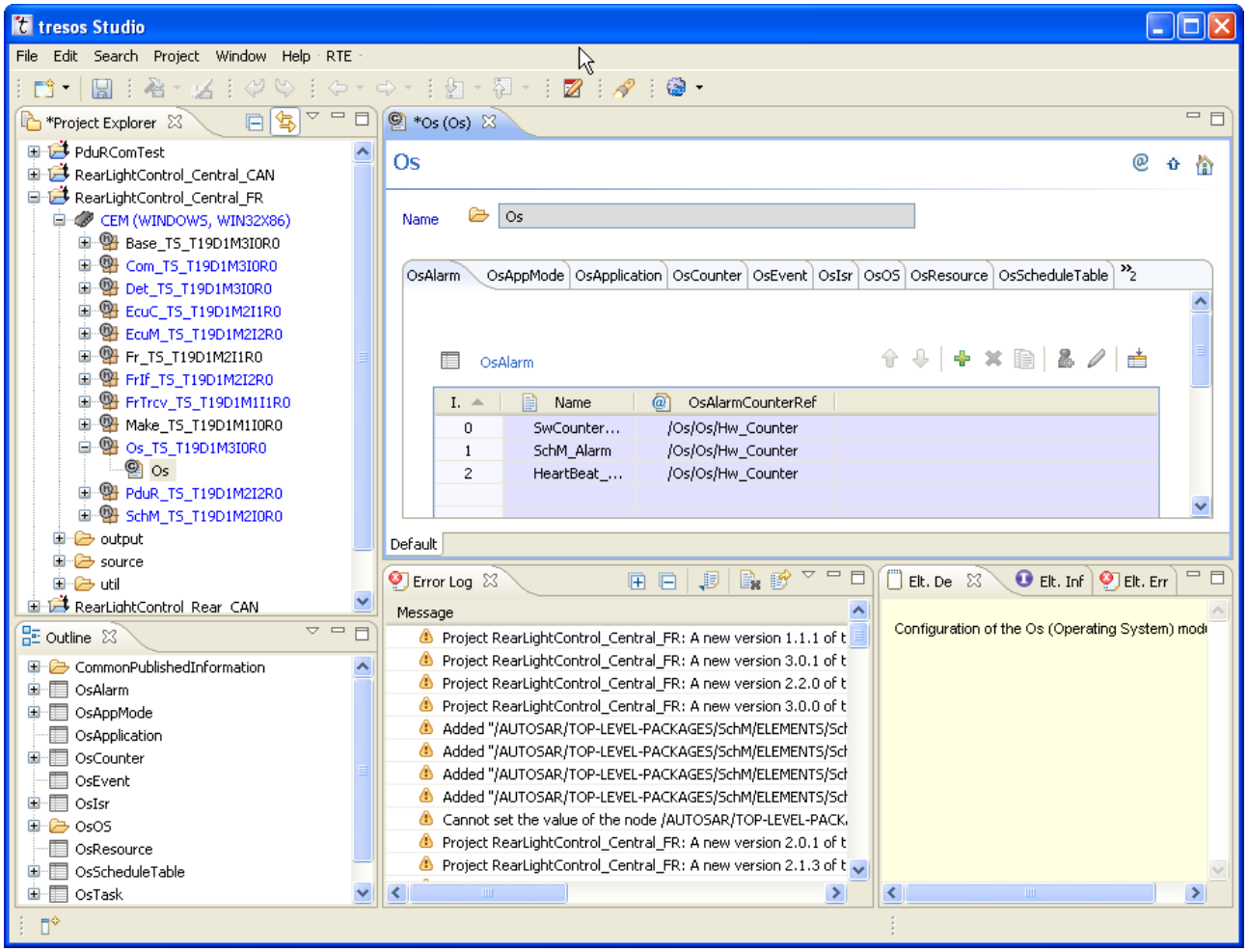


Figure 3.1. The main window of EB tresos Studio

3.3.2. Usage

Typical use scenarios for EB tresos Studio are:

- OEMs use EB tresos Studio for pre-integration of ECU standard-core software and deployment among their suppliers.
- Tier-one companies use EB tresos Studio for specification and code generation of basic software for their production ECU projects.
- Semiconductor vendors have chosen EB tresos Studio as the tool for the deployment of microcontroller drivers among their customers. Of course, EB uses EB tresos Studio as the configuration editor for all their EB tresos AutoCore products.

3.3.3. Benefits

You can immediatley experience the benefits of EB tresos Studio's Autosar methodology, when you work with the software tool:

- User-friendly configuration of basic software modules
- Competent implementation of the Autosar methodology
- Quick familiarization
- Open and documented interfaces
- Extendibility and interoperability

3.3.4. Features

EB tresos Studio offers a wealth of features, which simplify your job of configuring software modules for an *ECU*.

Furthermore, the tool offers interfaces and methods for tool-chain integration and customization:

- Generic configuration editor according to Autosar with many add-on features and improvements.
- Easy navigation within the ECU configuration data structure via
  - tabbed editors,
  - node views,
  - and clickable parameter references.
- Eclipse-based graphical user-interface, customizable via Java plug-ins.
- Validation of configuration parameters and immediate user feedback of detected issues.
- Importers for legacy formats such as
  - *FIBEX* ,
  - *DBC* ,
  - and *LDF*
- Integration of company-specific software modules due to an open Java API and code generator engines
- Interoperability with system architecture tools such as dSPACE `SystemDesk`.

In detail, these features include:

Eclipse based framework
: Using Eclipse 3.x as the foundation technology ensures major extensibility and support for a fast growing set of software development related tooling.

Configuration capabilities
: EB tresos Studio provides support for configuration editors that allow to edit the configuration of an embedded software module.

Generic configurator
: EB tresos Studio provides a generic configuration editor that uses the definition of configuration parameters and automatically displays a dialog-based graphical user interface (GUI).

Custom configurators
: EB tresos Studio can handle multiple editors per module. Editors can be specifically implemented in Java for a specific module.

Code generator capabilities
: EB tresos Studio provides support for code generators that allow to transform a configuration into source code that can be compiled for an embedded target platform.

Commandline interface
: EB tresos Studio provides a commandline interface for code-generation to run in batch process.

Multiple generators
: Each module may provide its own or even multiple generators.

Template-based generator
: EB tresos Studio provides a code-template based generic generator engine.

Project-based configuration
: EB tresos Studio is capable of managing multiple configuration projects where each project may represent the configuration for a whole ECU.

Multiple modules
: A project is capable to host multiple modules that share a common configuration database.

Multiple configurations
: A module can be configured multiple times inside one project. [1]

Cross module dependencies
: Cross module-dependencies within the configuration of one ECU can be expressed via XPath-expressions and are evaluated constantly during configuration.

GUI hints
: A parameter definition of a module can contain GUI hints such as an enable/disable expression, a HTML-description or layout hints to allow the generic configurator provide a more useable interface.

List in the configuration
: Lists of elements in the configuration are shown as a table with multiple sortable columns.

Navigatable references
: From references in the configuration that point to loaded parameters the GUI can be navigated easily to the referenced element.

[1] Not every module may support this option.

Importer/Exporter

EB tresos Studio provides a generic importer/exporter interface that allows to add importers and exporters for foreign formats such as OIL, DBC, DIL, Fibex and annotates imported data so that a re-import is possible.

ECU Configuration 2.0

EB tresos Studio allows to configure, import and export Autosar ECU Configuration 2.0 compatible modules.

ECU Configuration 2.1

EB tresos Studio allows to configure, import and export Autosar ECU Configuration 2.1 compatible modules.

ECU Configuration 3.0

EB tresos Studio allows to configure, import and export Autosar ECU Configuration 3.0 compatible modules.

4 EB tresos Studio user's guide

4.1. Overview

The EB tresos Studio user's guide provides information on operating the tool via the graphical user interface and also from the commandline.

If you have never worked with EB tresos Studio, we encourage you to start reading Section 4.2: Background knowledge before proceeding to the other chapters.

If you have worked with EB tresos Studio and are familiar with the concepts or you just need to look up something, please proceed to the task-oriented instructions.

- The chapter Section 4.2: Background knowledge includes information about file formats and basic concepts within EB tresos Studio. You will need this knowledge to understand the rest of the user's guide
- The chapter Section 4.3: Using EB tresos Studio for the first time covers the very first start of EB tresos Studio.
- If you are not familiar with the GUI, please take a look at the chapter Section 4.4: The GUI main window. This chapter shows all elements of the main window and explains their purpose.
- The chapter Section 4.5: Changing the EB tresos Studio setup describes how to get information about the current set-up of the tool and how to change the settings.
- If you want to create your first configuration project, or want to learn more about what you can do with projects, you should take a look at Section 4.6: Working with projects.
- The chapter Section 4.7: Working with ECU configurations describes how you can create a ECU configurations for an Autosar module and what you can do with it.
- The chapter Section 4.8: Upgrading projects and configurations covers upgrading to a newer Autosar version, or to a new version of a module.
- All tasks around editing an ECU configuration are descirbed in Section 4.9: Working with the ECU configuration editor.
- How source code can be generated out of an ECU configuration is explained in Section 4.10: Generating code for projects.
- The chapter Section 4.11: Importing and exporting provides information around the topic *importing configuration data:*
  - The chapter Section 4.11.2: Importing and exporting configuration data provides step-by-step instructions on how to import data in general.
  - The chapter Section 4.11.2.2: Importing and exporting configuration data from and to the Autosar format shows how to import or export project configuration data in/from the Autosar format.

- The chapter Section 4.11.2.5: Importing CAN configurations from the DBC format shows how to import data from DBC file format and explains the mapping of the DBC information to Autosar tags.
- Accordingly, the chapter Section 4.11.2.6: Importing from the LDF format shows you how to import LDF data.
- Section 4.11.2.4: Importing from the OIL format shows you how to import OIL files.

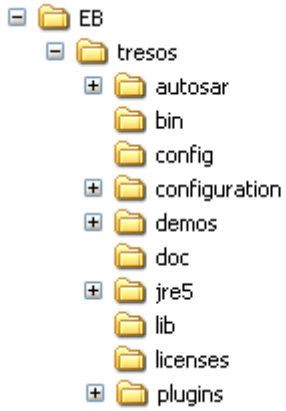
- EB tresos Studio also provides a command line mode, which is described in the chapter Section 4.12: Working with the command line. The command line mode includes two sub-modes:
  - project mode for already existing code and
  - legacy mode for projects that have not been started yet.

4.2. Background knowledge

The EB tresos Studio user's guide provides information on operating the tool via the graphical user interface and also from the commandline.

4.2.1. File and directory structure

A new EB tresos Studio installation has the following directory structure:



directory	description
`autosar`	contains the Autosar ECU configuration parameter definition for each supported Autosar version and a Perl script to generate EPD files from each parameter definition
`bin`	contains the executable files to start EB tresos Studio (`tresos_gui.exe` and `tresos_cmd.bat`)
`config`	used by the legacy generator

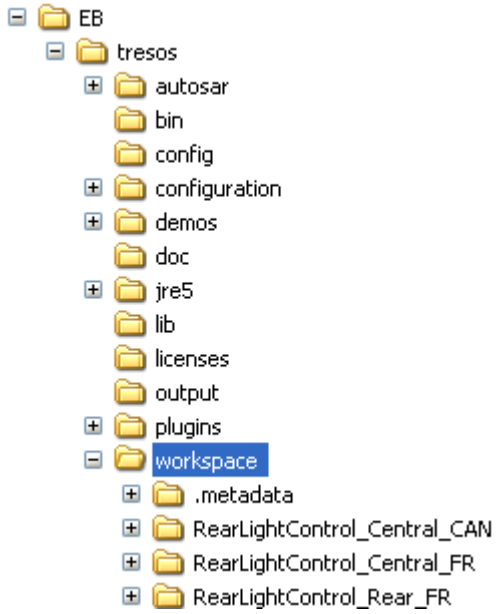
directory	description
`configuration`	runtime configuration information for EB tresos Studio like the plugin cache
`demos`	several demo plugins for developers which showcase the EB tresos Studio API
`doc`	program documentation like e.g. the User's Guide
`jre5`	Java runtime environment (required to run EB tresos Studio)
`lib`	program libraries internally used by the program
`licenses`	license agreements of third-party products
`plugin`	plugins and modules which are part of your EB tresos Studio installation

4.2.2. EB tresos Studio projects and workspaces

The EB tresos Studio tool enables you to work with multiple projects rather than a single configuration. Projects are arranged in a workspace.

4.2.2.1. The Workspace

A workspace is a folder by default in the installation tree of your EB tresos Studio installation. If the workspace folder is not present, it is automatically created upon the first start of the program. It is possible to have several workspaces managing different projects (see Section 4.5.4.1: Switching the workspace).



Besides projects the workspace stores non project specific preferences and also log filese in a meta data area.

EB tresos Studio always operates on one workspace which is initially selected on tool startup (see Section 4.5.4.1: Switching the workspace). The workspace can be switch via the **File** menu, EB tresos Studio restarts in this case with the new workspace open.

4.2.2.2. Projects

ECU configurations, system descriptions and also companion files like the source code of an application can be stored in a project. A project is represented as a folder with some hidden files describing the project meta data. The project folder normally resides beneath the workspace folder but can also be placed somewhere else in the filesystem. Projects can be imported into a workspace after which they show up in the EB tresos Studio GUI.

An EB tresos Studio project can host an ECU configuration consisting of BSW module configurations in the form of XML files or a system description model in the form of a database or both. ECU configurations and system descriptions are stored in a folder `config` benath the top level folder of the project. Arbitrary files can be created or imported into a project folder via the GUI of EB tresos Studio

4.2.2.3. File content types

Content types define the content of files more precisely than the file extension. This influences how the information read from files is intepreted or how the information will be saved to the files. In EB tresos Studio, content types are used in two locations: when using the Autosar importer or exporter (see Section 4.11.2.2: Importing and exporting configuration data from and to the Autosar format) and when using the command line (see Section 4.12: Working with the command line).

The content type of a file defines how a file is validated, e.g. against an xsd schema. The content type also determines how new files are written. Every input- and output file for each command on the command line may be equipped with a content type.

By default, EB tresos Studio knows two content types, `xdm` for the XDM file format and `asc` for Autosar files. In addition, parsers contained in other plugins or modules that you may have installed may define their own content types.

4.2.2.3.1. Content types for XDM files

Different versions of EB tresos Studio may use different versions of the XDM file format. Newer versions of EB tresos Studio are always able to read the older file versions. The following content types are available:

Content Type	Description
`xdm`	This content type does always refer to the latest version of the XDM file fomat (currently 3.0).

Content Type	Description
`xdm:2.0`	This content type refers to an older version of the XDM format which was used in versions prior to and including EB tresos Studio 2007.b.
`xdm:3.0`	This content type refers tothe version of the XDM format which is used since EB tresos Studio 2008.a.

4.2.2.3.2. Autosar content types

EB tresos Studio allows to import or export project configuration data from and into the Autosar format. To do so, you need to be aware of the Autosar content type of the file to be imported or exported. The term content type describes the fact that the Autosar specification has been released in several versions already, which are validated against different xsd schemas and provide differing characteristics.

Below is a list of all supported Autosar formats. To exchange files between different tools, you must use a format that can be checked by both tools when they use the same xsd file, which is provided by Autosar.

Content type	Description
`asc`	Default content type.
`asc:2.0`	Validates files against the Autosar xsd schema version 2.0.
`asc:spec2.0`	Validates files against the Autosar xsd schema version 2.0. Writes files as specified in the Autosar Specification of ECU Configuration version 2.0. **Differences to asc:2.0:** ► Empty values and non-configured values are allowed. This also applies to integer values. This means that empty value and value-ref tags may occur, e.g. `<value></value>`! However, `<value/>` may not occur. ► For configuration classes, the values are stored/accepted as defined within the specification: `PreCompile`, `Link`, `PostBuild`, `PostBuildLoadable`, and `PostBuildSelectable`. ► The order of xml elements is not specified.
`asc:2.1`	Validates files against the Autosar xsd schema version 2.1. This format enables exchanging files between tools if the tools use the same Autosar xsd file format/can check the filesâ€™ validity.

Content type	Description
`asc:3.0`	Validates files against the Autosar xsd schema version 3.0.

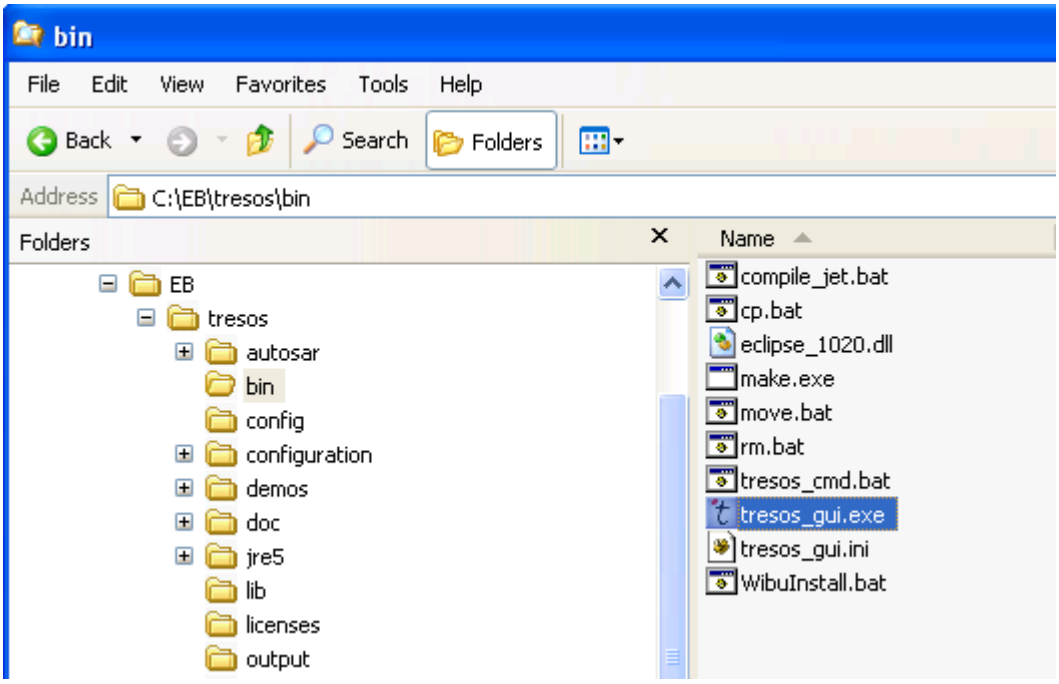
4.3. Using EB tresos Studio for the first time

4.3.1. Starting the graphical user interface

By default, the EB tresos Studio installer creates an icon on your desktop and adds an entry in the Windows Start menu. Both are labeled "tresos Studio". You can use either to start EB tresos Studio. The icon looks like this:

If you do not see such an icon:

- Locate the root directory of the EB tresos Studio installation `$TRESOS_BASE`.

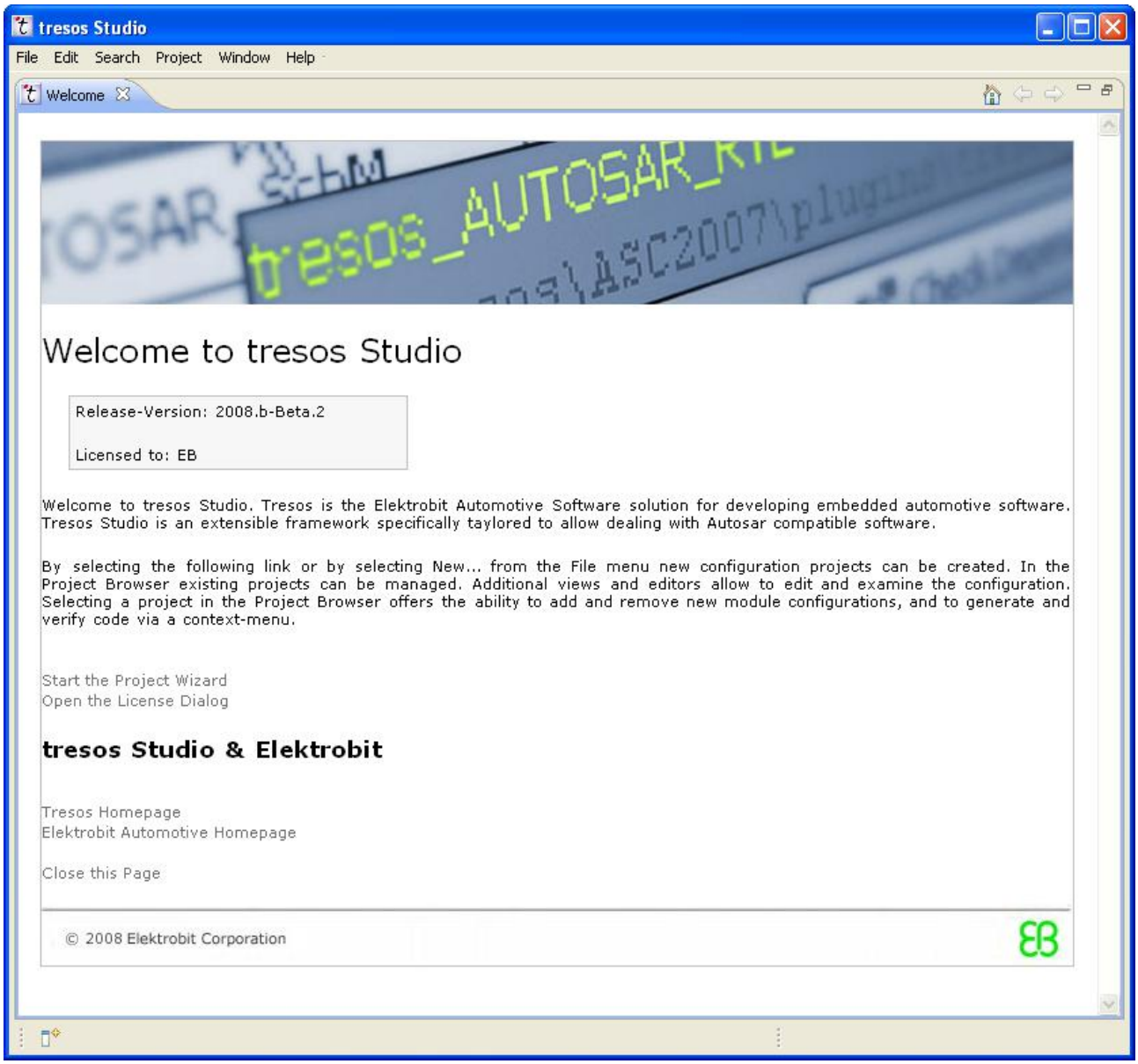


- In the `$TRESOS_BASE`, locate `bin\tresos_gui.exe`.
- Double-click `bin\tresos_gui.exe`.

  The GUI is being started.

4.3.2. First launch

If you start EB tresos Studio for the first time, the graphical user interface starts up with a "Welcome" view:



On the Welcome view you can find the version of the EB tresos Studio release you are using and the name of the licensee (if a license is already installed). The view also displays some hints for first-time users and gives shortcuts to common functions like create a new project or opening the license dialog.

To start working with EB tresos Studio you have to close the Welcome view. To do so, click on the "Close this page" link at the bottom of the page, or click the "x" icon next to the "Welcome" title. If you want to open the view again, choose the "Welcome" item from the "Help" menu.

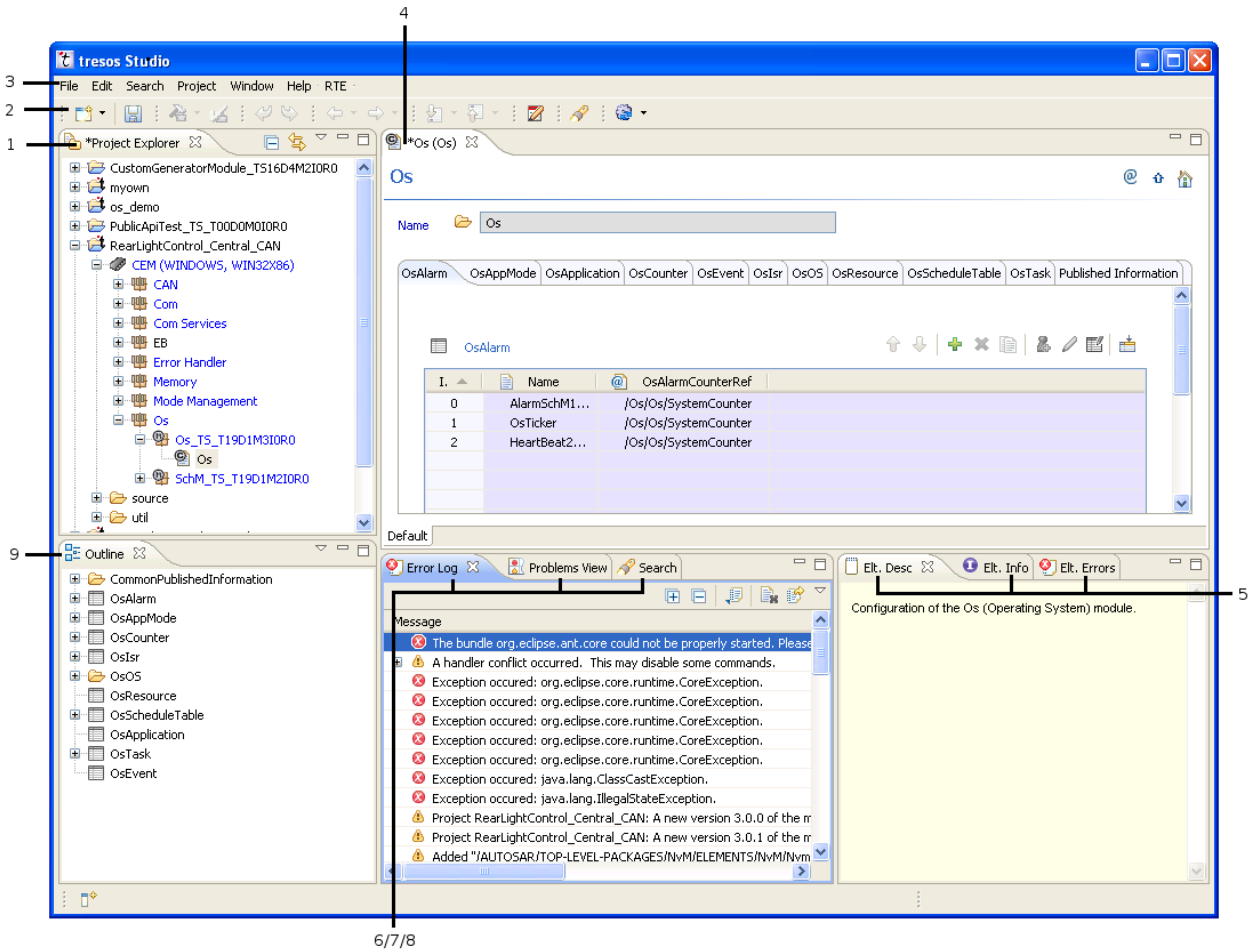
After closing the view, you see the EB tresos Studio main window. The contents of this window will be explained in Section 4.4: The GUI main window.

4.4. The GUI main window

This chapter is going to introduce you to the main window of EB tresos Studio. You will get to know how the main window is organized and what functions are available through the menus and the toolbars.

4.4.1. Overview

The graphical user interface (GUI) of EB tresos Studio consist of views, e.g. the *Project Explorer*:

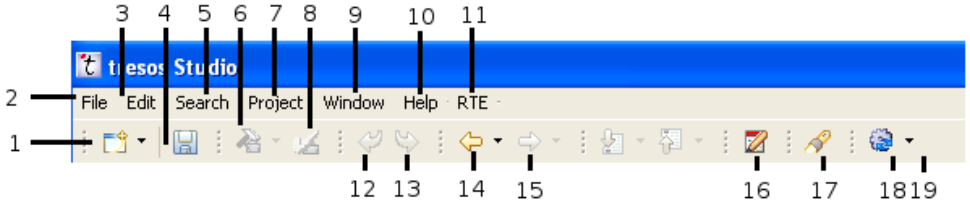


Number	Window	Description
1	*Project Explorer*	The *Project Explorer* displays all EB tresos Studio configuration projects in a workspace (see Section 4.2.2: EB tresos Studio projects and workspaces) and allows you to edit projects and open *Editor* views. The default location of the *Project Explorer* view is on the left of the screen.

Number	Window	Description
2	Toolbar	The toolbar is located in the upper left corner of the screen.
3	Menubar	The menubar is located in the upper left corner of the screen. In addition to the default menu items, there may additional menu items present. These are contributed by the additional plug-ins and modules you have installed and should be documented there.
4	*Editor* view	The *Editor* view is in the center of the EB tresos Studio main window; it enables editing the configuration data of a project.
5	*Element Outline* view	The *Element Outline* view is on the right bottom of the workbench; it consists of multiple views that show information about the currently selected parameter in the *Editor* view.
6	*Problems* view	The *Problems* view displays the messages found in the configuration of a loaded project. The default location of the *Problems* view is on the bottom of the screen.
7	*Error Log* view	The *Error Log* view is the place in the GUI where all errors are logged. The *Error Log* receives error messages from EB tresos Studio components and from other Eclipse components. The list of messages in the *Error Log* is preserved in between the sessions of EB tresos Studio. The default location of the *Error Log* view is on the bottom of the screen.
8	*Search* view	The *Search* view enables searching and finding files, specific configuration parameters, or text strings.
9	*Node Outline* view	The *Node Outline* view on the left bottom of the workbench, which shows a tree view of the editable configuration parameters that you may edit in the *Editor* view.

4.4.2. Menubar and Toolbar

The menubar and the toolbar are located in the upper left corner of the screen:

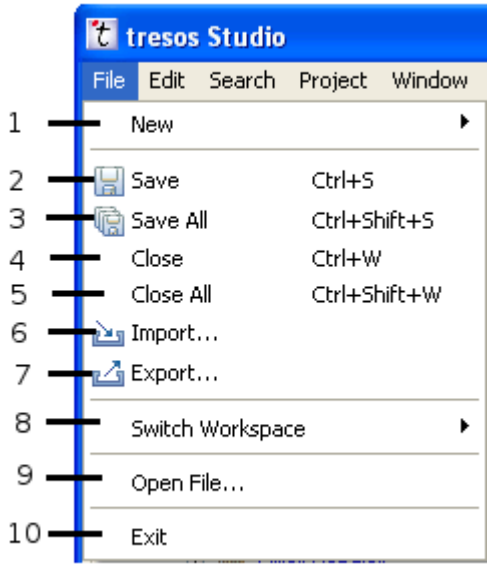


Number	Menu	Description
1	*New project wizard* icon	Select to open a new project wizard.
2	*File* menu	Select for project unspecific tasks.
3	*Edit* menu	Select for tasks related to editing project data.
4	*Save configuration project* icon	Select to save the configuration project of the currently active editor window (see Section 4.6.2: Saving a project).
5	*Search* menu	Select to search for and to find files, specific configuration parameters, or text strings
6	*Generate code for the currently selected project* icon	Select to start generating code, see details in Section 4.10: Generating code for projects.
7	*Project* menu	Select to verify and generate code, see details in Section 4.10: Generating code for projects.
8	*Verify selected project* icon	Select to start verifying code, see details in Section 4.10.1: Verifying a project
9	*Window* menu	Select the menu to set preferences and to choose views/windows, see details in Section 4.6.6: Closing and opening projects.
10	*Help* menu	Select to access all global information and help functions for EB tresos Studio, see details in Section 4.5.2: Viewing setup information.
11	*RTE* menu	This item is only available if you have an *Rte* module installed. It enables you to start the *Contract Phase*. You will find more information about this item in the user manual of the Rte module.
12	*Undo* icon	Select to undo the last edit. The button is active only if undoable change options are available.
13	*Redo* icon	Select to redo the last undone edit. The button is active only if redo change options are available.
14	*Back* icon	Select to move to the last editing position.

Number	Menu	Description
15	*Forward* icon	Select to move forward to the next editing position. This option is only available if the **Back** button or the corresponding menu was selected previously.
16	*RTE Contract Phase generation* icon	This item is only available if you have an *Rte* module installed. Select to open the *Please select Software Component Description files* window to start the RTE Contract Phase. You will find more information about this item in the user manual of the Rte module.
17	*Search* icon	Select to open the *Search* window default page, which is currently the *File Search* page.
18	*Run all enabled autoconfig wizards* icon	Select to run one or all enabled *Autoconfigure Wizard*(s) for a specified project.
19	*Arrow down* icon	Select to configure autoconfiguration wizards, calculate default values, or to calculate handle-IDs.

4.4.2.1. File menu

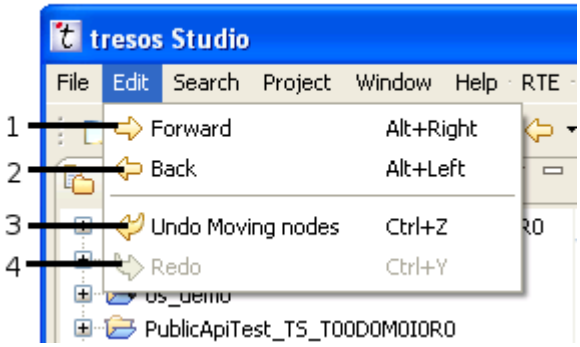
The *File* menu is located on the far left of the EB tresos Studio main window.



Number	Menu	Description
1	*New*	Select *Project* in the sub menu to create a new EB tresos Studio project Section 4.6.1: Creating a new project.
2	*Save*	Select to save the content of the currently active editor. If the editor is a configuration editor, the associated configuration project is saved Section 4.6.2: Saving a project.
3	*Save all*	Select to save the content of all open editors.
4	*Close*	Select to close the currently active editor. If you have modified any content since saving, a pop-up window is going to ask you to save unsaved content (see Section 4.6.6.1: Closing projects).
5	*Close all*	Select to close all open editors. If content of one or more editors is unsaved, a pop-up window is going to ask you to save unsaved content.
6	*Import*	Select to import projects (see Section 4.11.1.1: Importing a project).
7	*Export*	Select to export projects (see Section 4.11.1.2: Exporting a project).
8	*Switch Workspace*	Select to select a new workspace directory. If you select a new workspace directory, EB tresos Studio shuts down and restarts with the new workspace. See the concept of workspaces Section 4.2.2: EB tresos Studio projects and workspaces.
9	*Open File*	Select to open a file.
10	*Exit*	Select to close EB tresos Studio. A pop-up window is going to ask you if you want to save any changes before closing the tool.

4.4.2.2. Edit menu

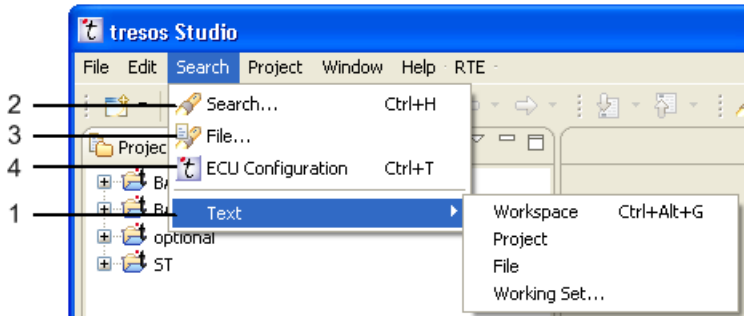
The *Edit* menu is the second drop down menu from the left. The menu items in this menu are active only if you are working with an editor view.



Number	Menu	Description
1	*Foward*	Select to move to the next editing position in the editor.
2	*Back*	Select to move to the previous editing position you worked on in the editor.
3	*Undo ...*	Select to undo the last change. Instead of "..." the name of the action that would be undone is displayed.
4	*Redo ...*	Select to redo the last change. Instead of "..." the name of the action that would be redone is displayed.

4.4.2.3. Search menu

The *Search* menu enables you to search for and to find files, specific configuration parameters, or text strings.

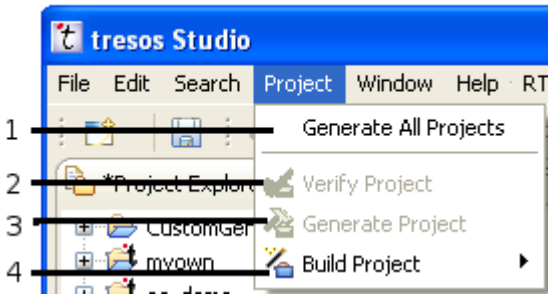


Number	Menu	Description
1	*Text*	Select to search for text strings. As search-pattern, the currently selected text is used. This menu option will open a submenu with different search scope options.

Number	Menu	Description
2	*Search*	Select to open the *Search* window, which is a dialog with a tab for every different type of search. If this dialog is invoked from a configuration editor, the *Tresos Search* is active directly when the dialog is opened.
3	*File*	Select to open the file search. Within the file search you may search for files wich a specific name or content.
4	*ECU Configuration*	Select to open the configuration parameter search within open EB tresos Studio projects. Instead of clicking the menu item, an alternate way of opening the dialog is to press **Ctrl-T**.

4.4.2.4. Project menu

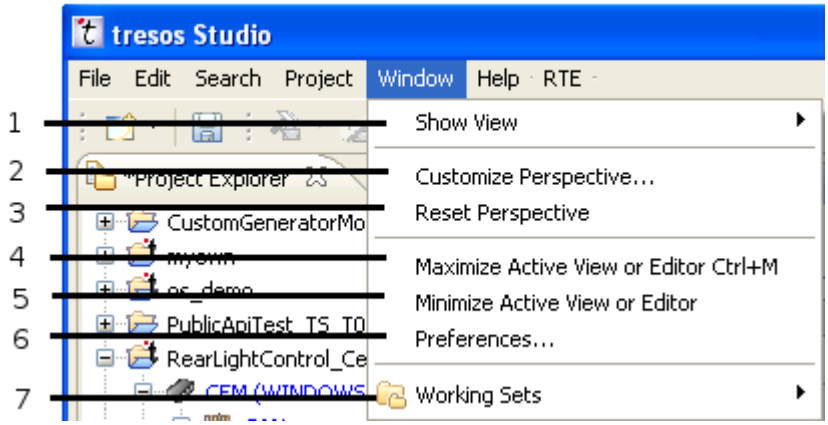
The *Project* menu is the fourth drop down menu from the left.



Number	Menu	Description
1	*Generate All Projects*	Select to generate the code for all projects Section 4.10.2: Generating a project.
2	*Verify Project*	Select to verify the project currently selected in the Project Explorer Section 4.10.1: Verifying a project. If no project is selected in the *Project Explorer*, this entry is grayed out.
3	*Generate Project*	Select to generate the project that is currently selected in the *Project Explorer* (see Section 4.10.2: Generating a project). If no project is selected in the *Project Explorer*, this entry is grayed out.
4	*Build Project*	Select to open a submenu that contains a list of generation modes Section 4.10.3: Building a project. The available generation modes are determined by the modules of the currently selected project. Please refer to the documentation of your installed modules. If no project is selected in the *Project Explorer*, this entry is grayed out.

4.4.2.5. Window menu

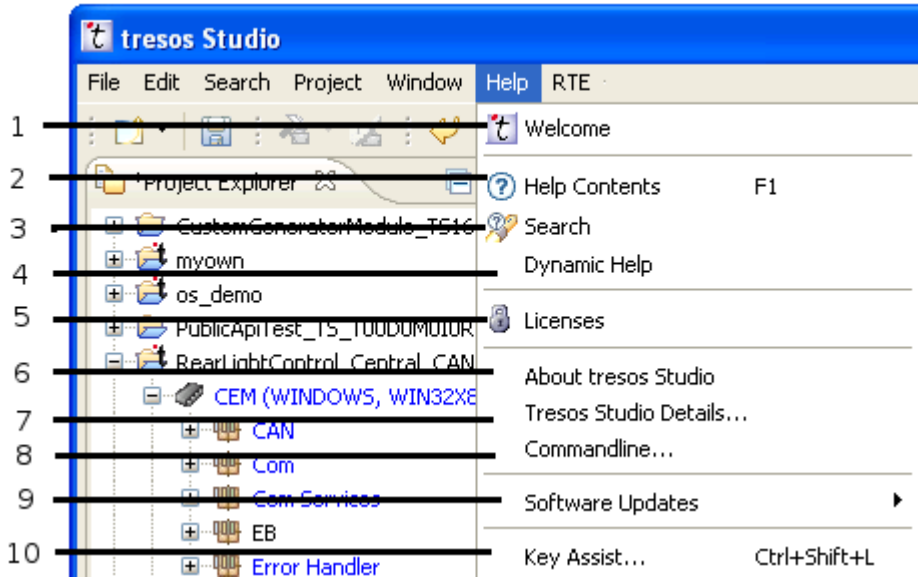
The *Window* menu is the fifth drop down menu from the left.



Number	Menu	Description
1	*Show View*	Select *Show View*, then select *Other* to reopen a view. A view in this context is a part of a window such as the *Project Explorer*.
2	*Customize Perspective*	Curently not used by EB tresos Studio.
3	*Reset Perspective*	Select to reset all views/windows to their default position and settings.
4	*Maximize Active View or Editor*	Select to enlarge the currently selected view to its maximum. This means that all other views will be minimized and may disappear from view.
5	*Minimize Active View or Editor*	Select to minimize the currently active view. This means that the active view may disappear from view and just the other views are displayed.
6	*Preferences*	Select to change the EB tresos Studio preferences Section 4.5.3: Setting preferences.
7	*Working Sets*	Curently not used by EB tresos Studio.

4.4.2.6. Help menu

The *Help* menu provides information about the tool, software updates, and licenses.

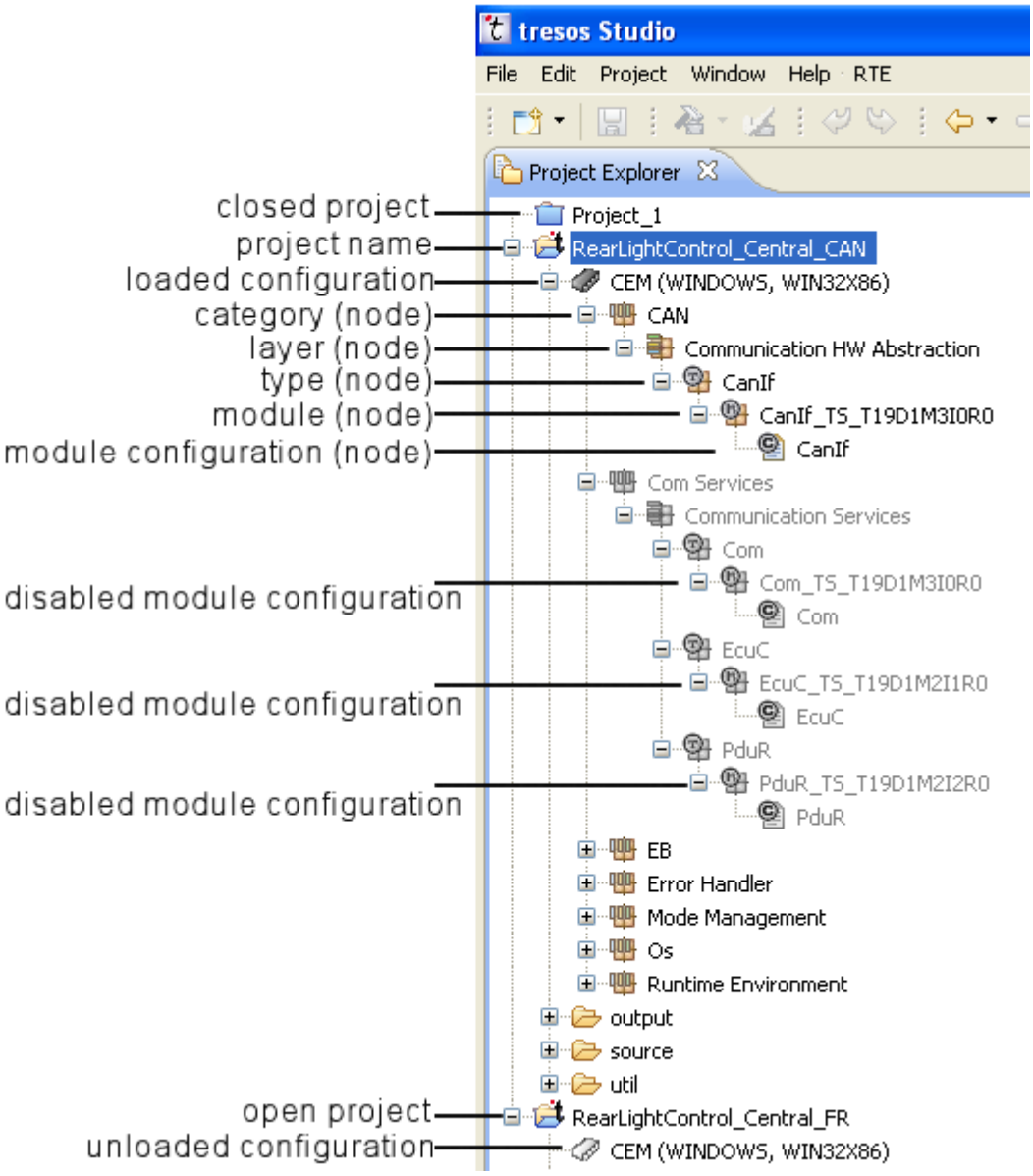


Number	Menu option	Description
1	*Welcome*	Select to reopen the welcome screen.
2	*Help Contents*	Select to open the help window and browse the online help function of EB tresos Studio.
3	*Search*	Select to search the online help of EB tresos Studio.
4	*Dynamic Help*	Currently not supported in EB tresos Studio.
5	*Licenses*	Select to open the license dialog (see Section 4.5.1: Managing licenses).
6	*About tresos Studio*	Select to access information about the EB tresos version number, build number, and features.
7	*Tresos Studio Details*	Select to access detailed information about EB tresos Studio, to list all modules, and all bundles.
8	*Commandline*	Select to browse the EB tresos Studio commandline options and view the commandline options help messages.
9	*Software Updates*	Currently not supported in EB tresos Studio.
10	*Key Assist*	Currently not supported in EB tresos Studio.

4.4.3. Project Explorer view

The *Project Explorer* is the main utility to work with EB tresos Studio projects. The *Project Explorer* displays all EB tresos Studio configuration projects in a workspace and allows you to edit projects and open Editor views.

The project explorer displays a tree. The tree starts with the projects on the first level. The second level contains the ECU configuration node. Beneath that level, the tree continues with several layers down to the module configurations. All items of the tree are represented by different icons. The items may change their state which is also reflected by their icon. The following picture gives you a comprehensive overview:



Note

A project which contains an ECU configuration usually is called a *configuration project* for short.

4.4.3.1. Project and ECU configuration states

Projects and ECU configurations can be in different states each. A project can be either open or closed. An ECU configuration can either be loaded or unloaded.

The following table gives an overview over the different project states and explains their use:

Project State	Icon	Description
open		In this state the *Project Explorer* displays the ECU configuration node for this project. This is the default state for projects after start-up.
closed		In a closed project you cannot edit any configurations, nor can you generate any code. The project consumes the least possbile resources.

The following table gives an overview over the different ECU configuration states and explains their use:

ECU Configuration State	Icon	Description
unloaded		In an unloaded ECU configuration you can view the configuration's properties such as the ECU Id or the target and derivate values, but you cannot edit them. The EUC configuration data is not loaded and therefore you can neither edit configurations, nor can you generate any code in this state. The benefit of this state is its very low resource consumption. This state is the default state for projects when EB tresos Studio is started.
loaded		A loaded ECU configuration is fully functional. Loaded configurations can be edited, modules can be added and code can be generated. Beneath the ECU configuration node, a hierarchical configuration tree displays configurations grouped according to categories. Newly created projects are in this state.

4.4.3.2. Layering and grouping of module configurations

The *Project Explorer* groups module configurations according to the modules they belong to. The modules themselves are grouped into hierarchical layers of categories.

Layer name	Icon	Description
Loaded configuration	icon in the shape of a chip	Represents the loaded configuration of a project.
Category node/layer	icon contains three vertically ordered bars	Provides a horizontal grouping of modules within an operating system stack.

Layer name	Icon	Description
Layer node/layer	icon contains three horizontally ordered bars	Provides a vertical grouping of modules within an operating system stack.
Type node/layer	icon with a *T* on the folder	Describes the functionality of the module, e.g. Can for a CAN driver.
Module node/layer	icon with an *M* on the folder	Is the actual implementation of a module, such as a CAN driver for a specific target and derivate.
Module configuration node/layer	icon with a *C* on the folder	Is the configuration for an actual implementation of a module (see module node); the leaf node represents either: ► a module configuration or ► an editor view if the module provides multiple editor views for configuration.

4.4.3.3. Changing the visible layers

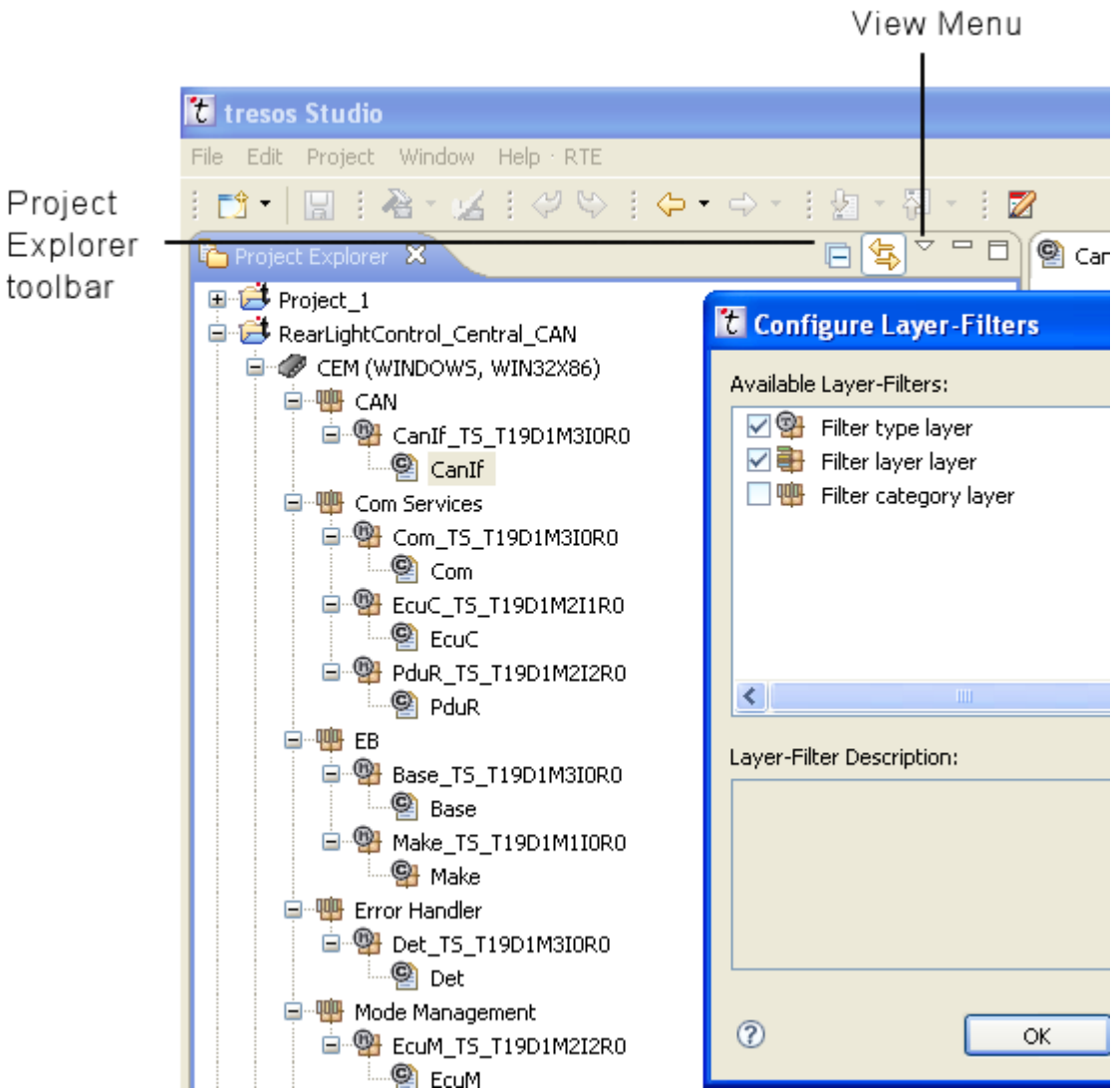
You can customize the layer grouping of the *Project Explorer* by selecting which layers shall be displayed. To change this:

► Select the **View Menu** button in the Project Explorer toolbar.

The **View Menu** menu opens up.

► Select the *Layer Filters* menu option.

The *Configure Layer Filters* window opens up.



If a layer is disabled, it is excluded from the tree. The content beneath the hidden nodes however, is not hidden. Instead, the content of the hidden nodes is displayed one level higher than it normally would have been. Nodes are grouped together if their parents are the same and if they share the same label.

The *Configure Layer Filters* window provides checkboxes to disable layers.

To disable the *category* layer

- Select the box *Filter category layer*

To disable the *layer* layer

- Select the box *Filter layer layer*

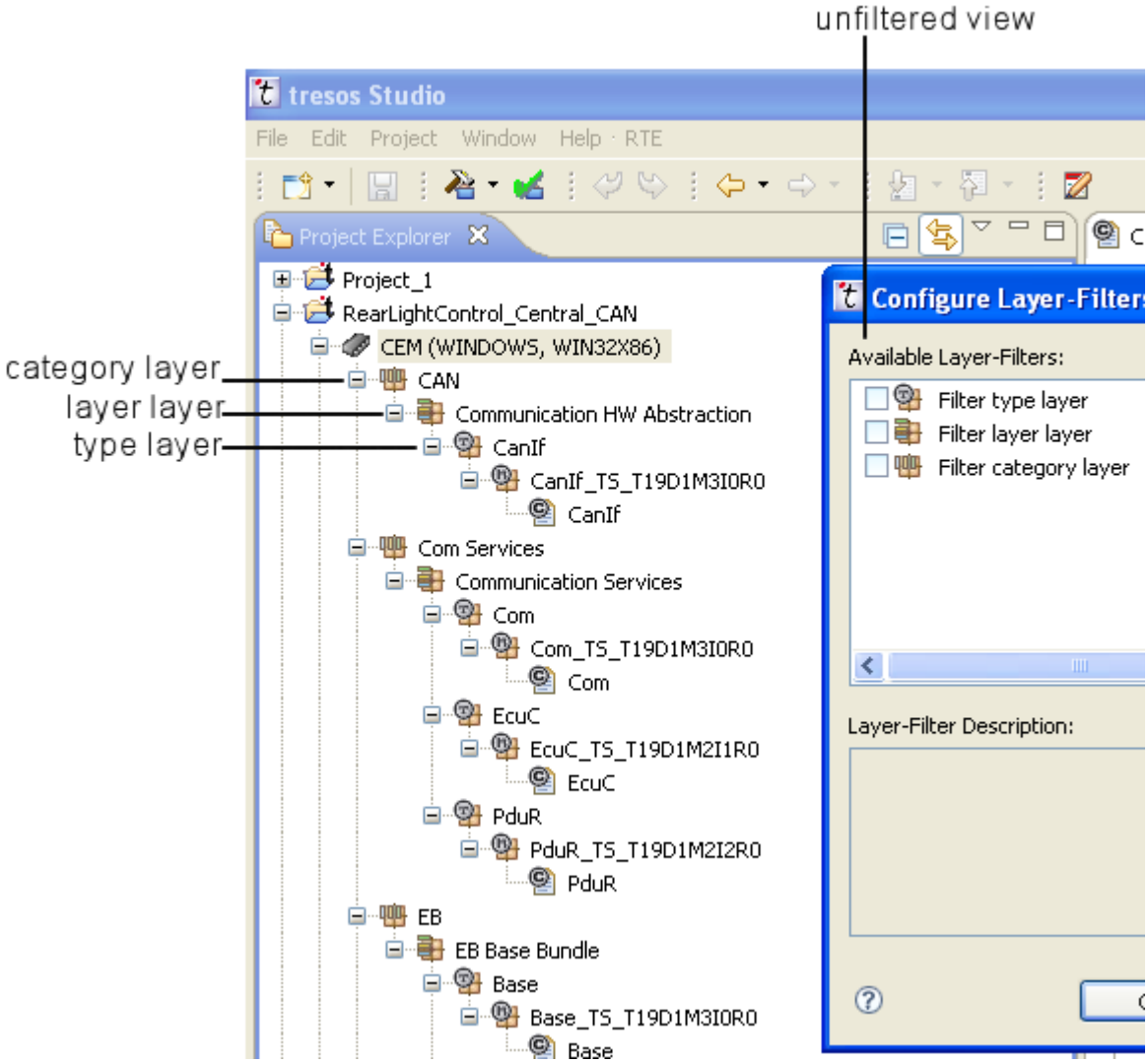
To disable the *type* layer

- Select the box *Filter type layer*

Example: Unfiltered view

- The unfiltered view shows
  - the category layer
  - the layer layer
  - the type layer

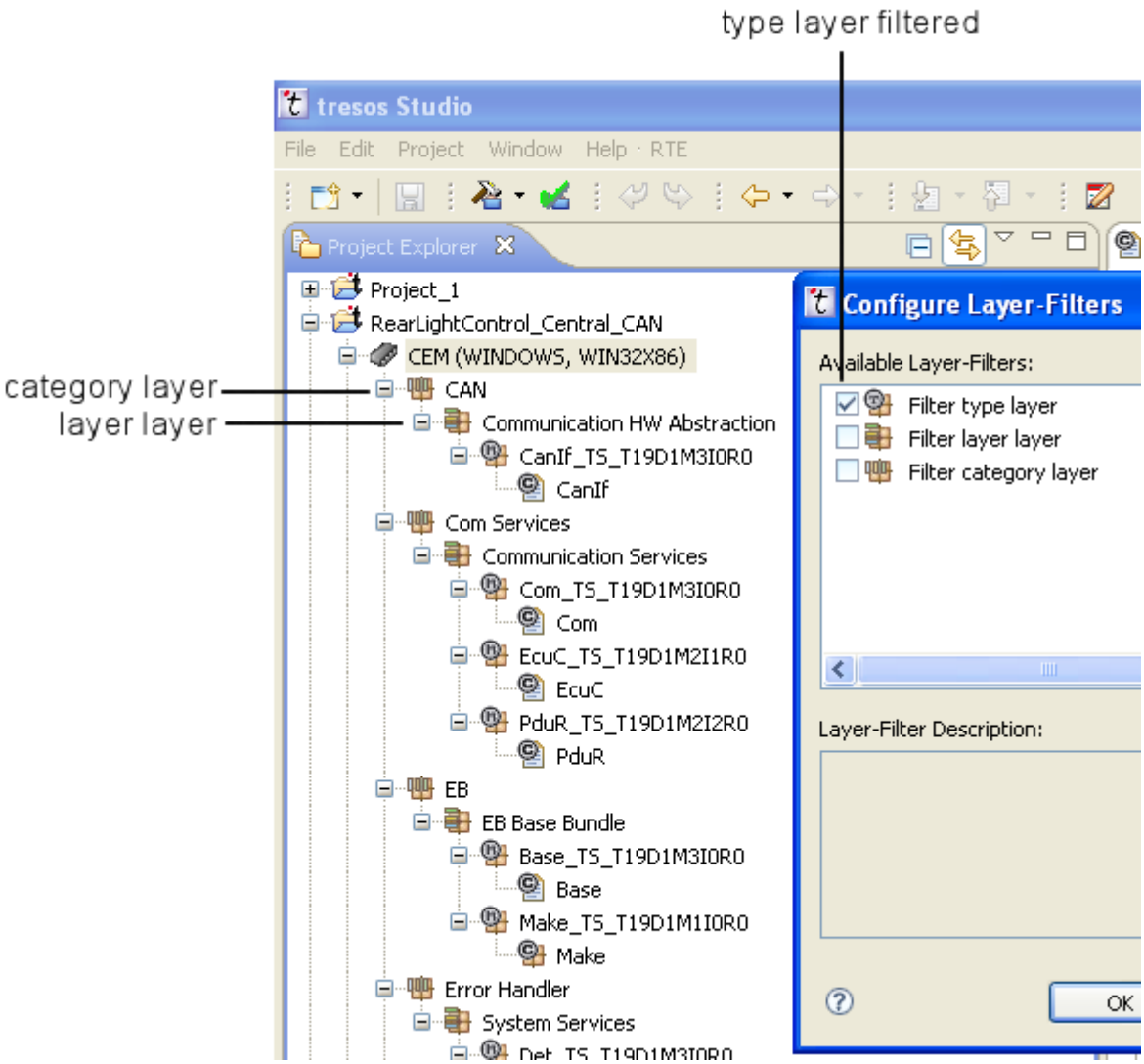
  as well as the module layer and the module configuration layer



Example: Type layer filtered out

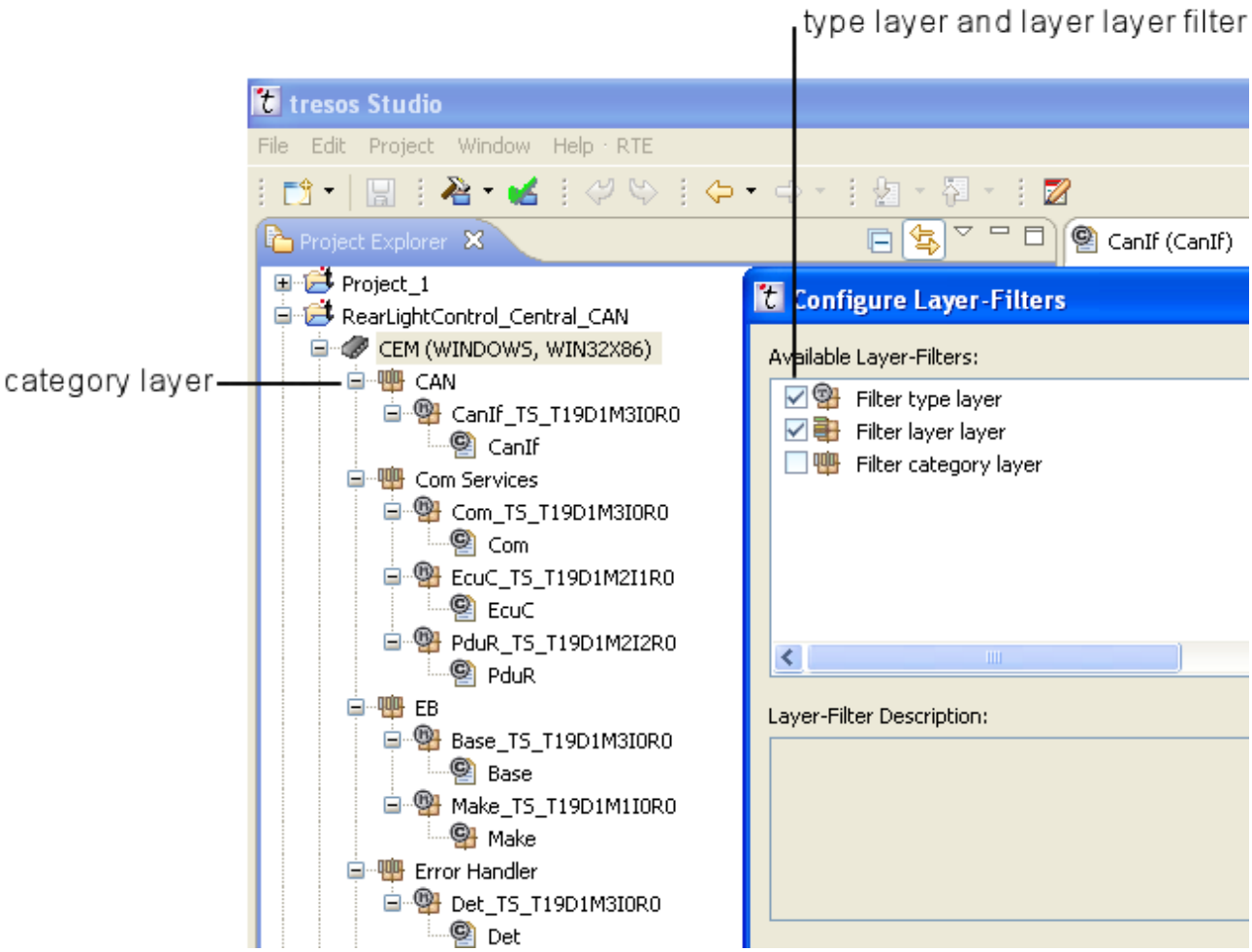
- If the type layer is filtered out, the *Project Explorer* shows
  - the category layer

- the layer layer
- as well as the module layer and the module configuration layer



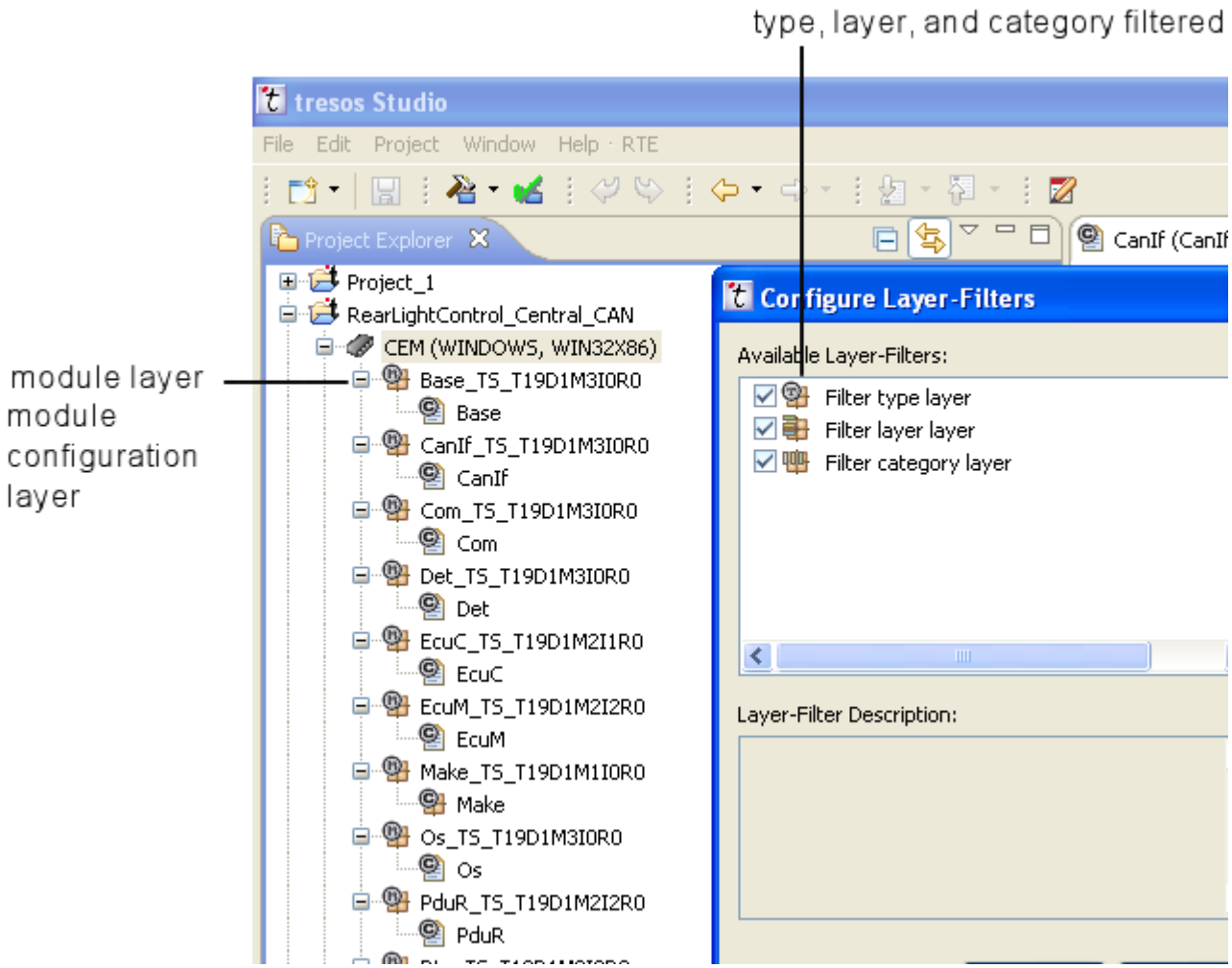
Example: Type and layer layer filtered out

- If the type and layer layer is filtered out, the *Project Explorer* shows
  - the category layer
  - as well as the module layer and the module configuration layer



Example: Type, layer, and category layer filtered out

- If the type, layer, and category layer are filtered out, the *Project Explorer* shows the only module layer and the module configuration layer.

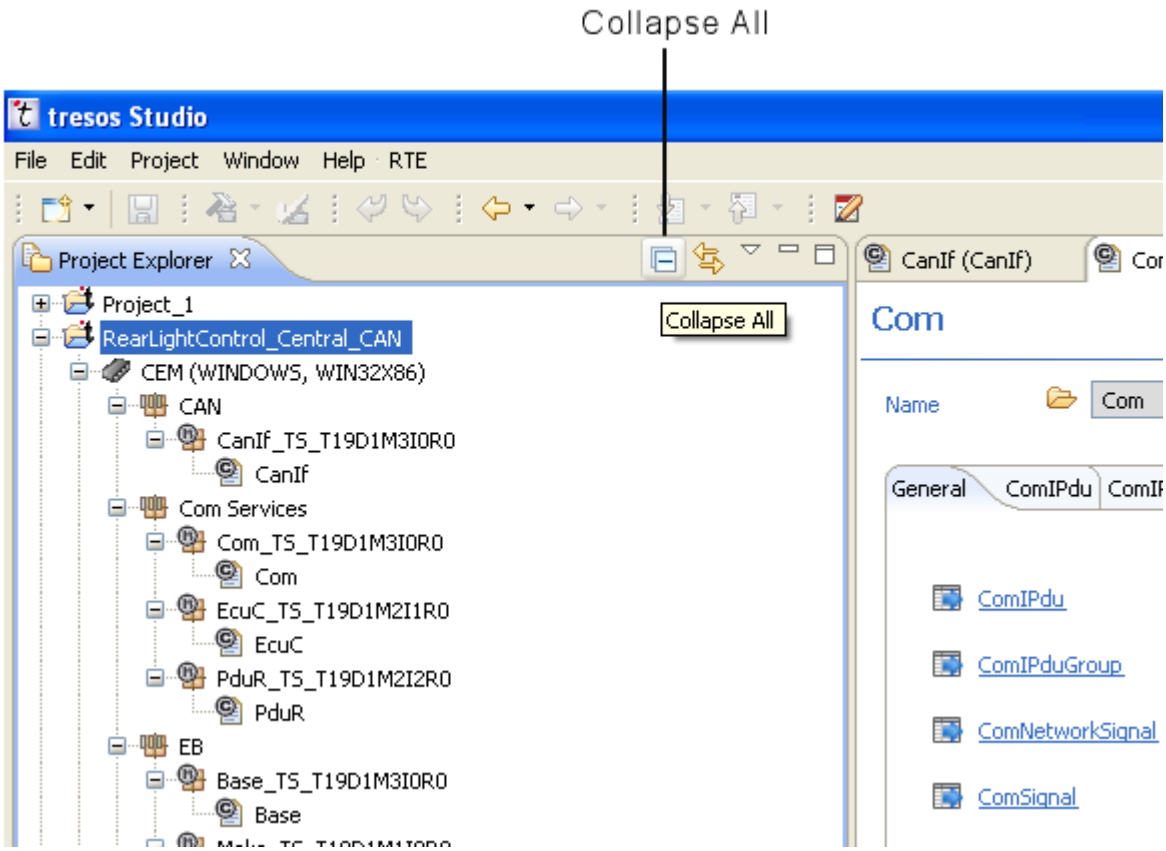


4.4.3.4. Collapsing the Project Explorer tree

To collapse the whole project tree inside the *Project Explorer* so that only the project names are visible:

- Select the *Collapse All* menu option from the *Project Explorer* toolbar.

  Only project names are visible now.

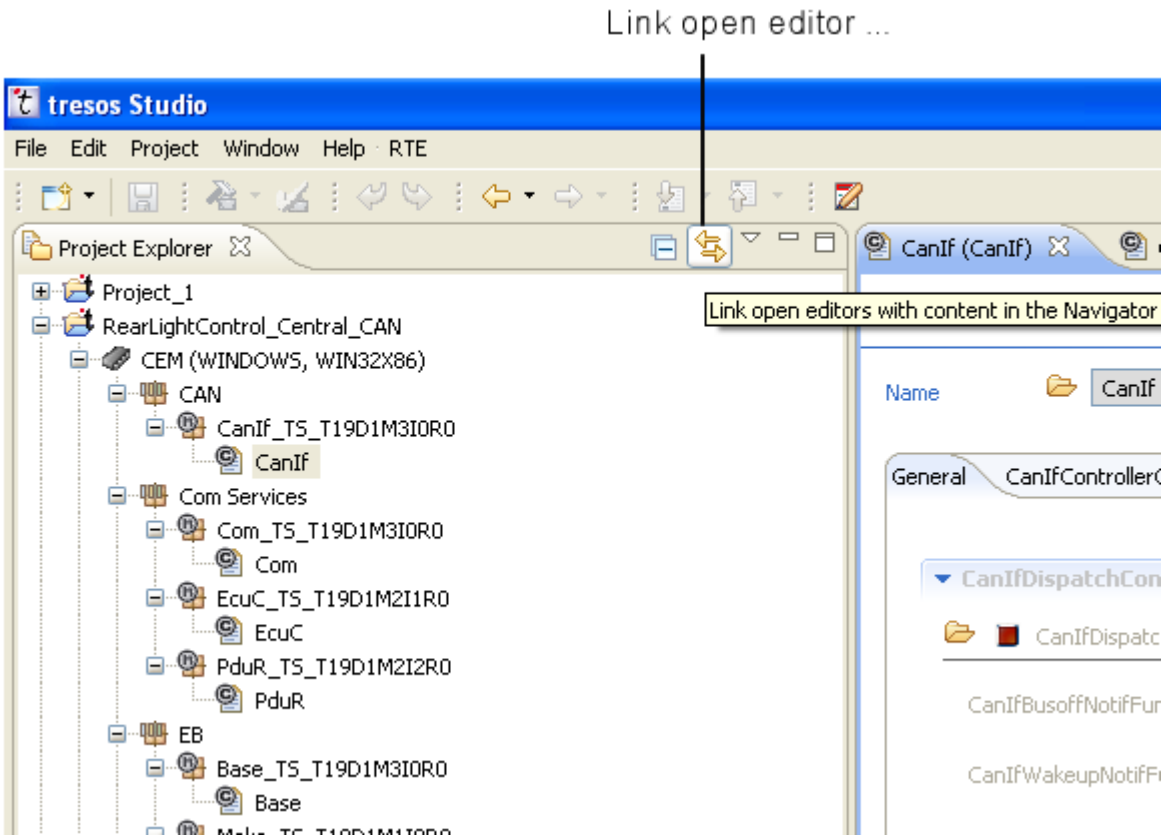


4.4.3.5. Synchronizing the Project Explorer with the Editor view

To synchronize the *Project Explorer* view with the Editor view on the right:

- Select the **Link open editors with content in the Navigator** toggle switch.

  The *Project Explorer* expands a module configuration and scrolls it into view for which ever corresponding editor view is selected.



4.4.3.6. Indicating different project states with colors

The *Project Explorer* uses colors to indicate the different states that the module configuration can be in. To change colors of any or all meanings, please refer to Section 4.5.3.6: Setting colors and fonts .

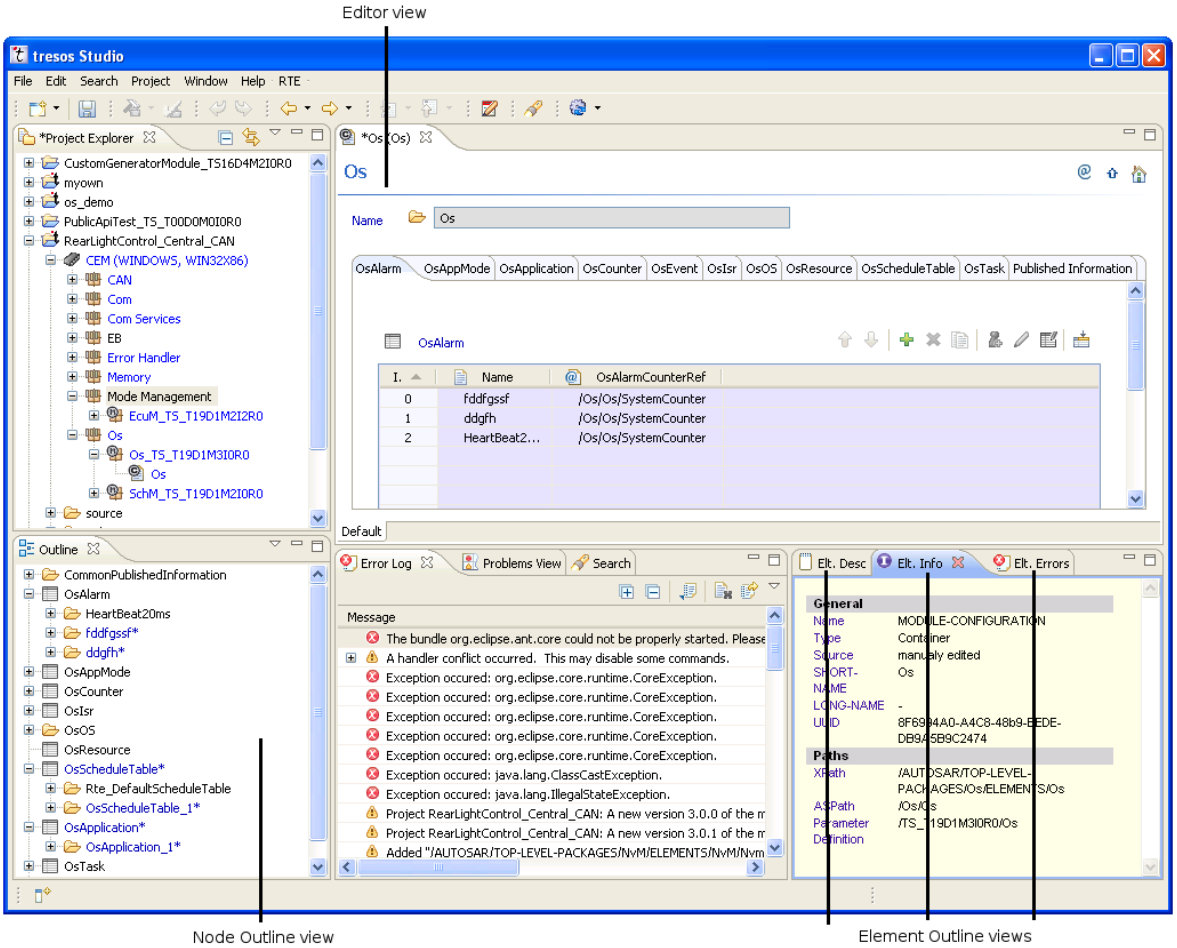
Color	Meaning
Black	default color for nodes
Blue	▶ default color for modified and not yet saved module configurations ▶ used in layers in helping to find the modified module configuration if the *Project Explorer* tree is collapsed ▶ used in editor views to highlight modified parameters
Gray	▶ default color for disabled module configurations ▶ if one or more module configurations are disabled, the whole subtree of the disabled module configurations is shown as disabled in gray

4.4.4. Configuration Editor

EB tresos Studio provides a standard implementation of a generic configuration editor that takes the parameter definition of a module as input and creates a dialog-based GUI.

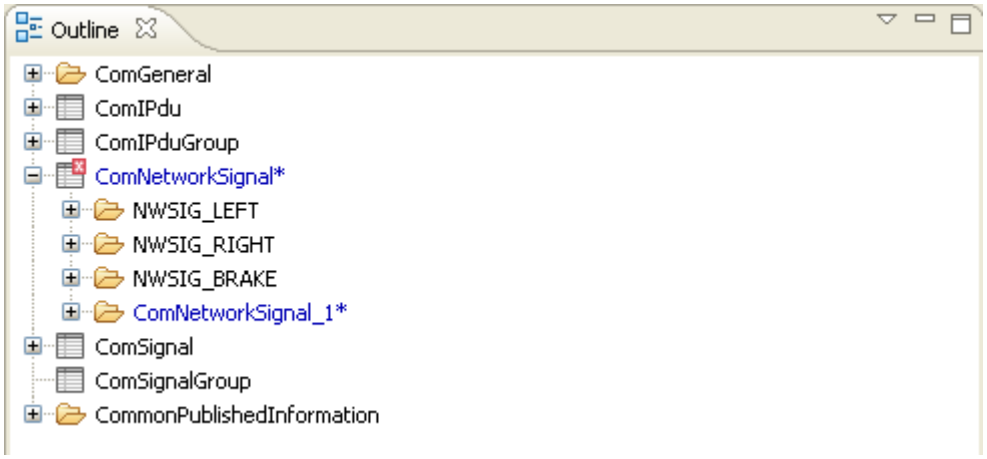
An editor consists of:

- the *Editor* view in the center of the EB tresos Studio window, which enables editing the configuration data of a project
- the *Node Outline* view on the left bottom of the workbench, which shows a tree view of the editable configuration parameters that you may edit in the *Editor* view
- the *Element Outline* views on the right bottom of the workbench, which consists of multiple views that show information about the currently selected parameter in the *Editor* view



4.4.4.1. Node Outline view

The *Node Outline* is a tree view outline of the editor contents. The *Node Outline* shows the editor contents in a hierarchical structure. Its main purpose is to navigate between parameters. The *Node Outline* displays the same icons as the Editor view for parameters.



To filter the displayed items in the *Node Outline*:

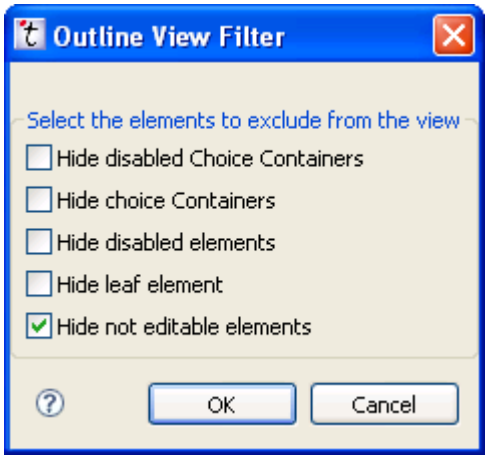
- Select the *Menu* symbolized by a small arrow down.

  A context menu opens up.

- Select *Filters*.

  The *Outline View Filter* window opens up.

Check the checkboxes according to the results you wish to obtain:



- To display just the active elements of a choice, check **Hide disabled Choice Containers**.
- To hide containers inside a choice element, click **Hide choice Containers**.
- To hide grayed out elements, check **Hide disabled elements** .

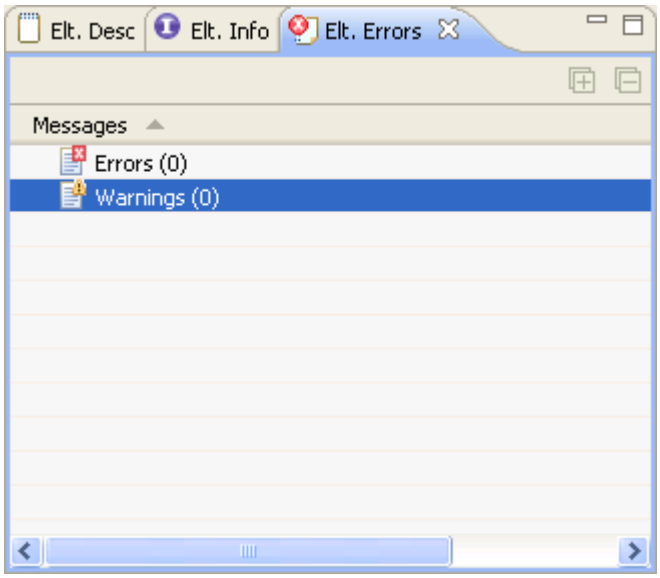
- To hide parameters and references that show just the structure of the pages in the editor, check **Hide leaf element**.
- To hide elements that cannot be changed in the editor, such as Autosar CommonPublishedInformation, check **Hide not editable elements**.

4.4.4.2. Element Outline views

The *Element Outline* consists of three views that display information about the parameter currently selected in the *Editor* view. The three views are:

- the *Element Errors* view
- the *Element Info* view
- the *Element Description* view

4.4.4.2.1. Element Errors view

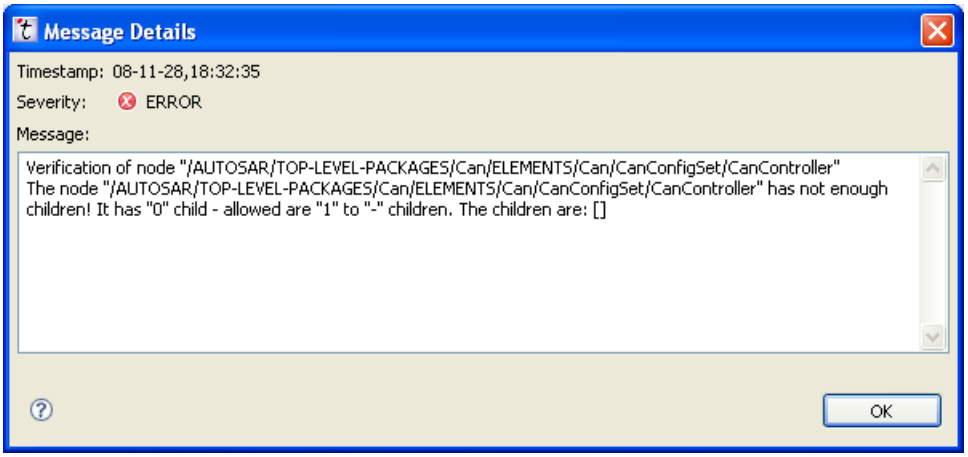


The *Element Errors* view displays messages and errors just as the *Problems* view (see Section 4.4.6: Problems view) but only for the parameter currently selected in the *Editor* view. The error code provided is a tool-internal code used by `EB tresos Studio` developers to locate an error in the source code of the tool.

To see detailed message information:

- Double-click on the message in the *Element Errors* view.

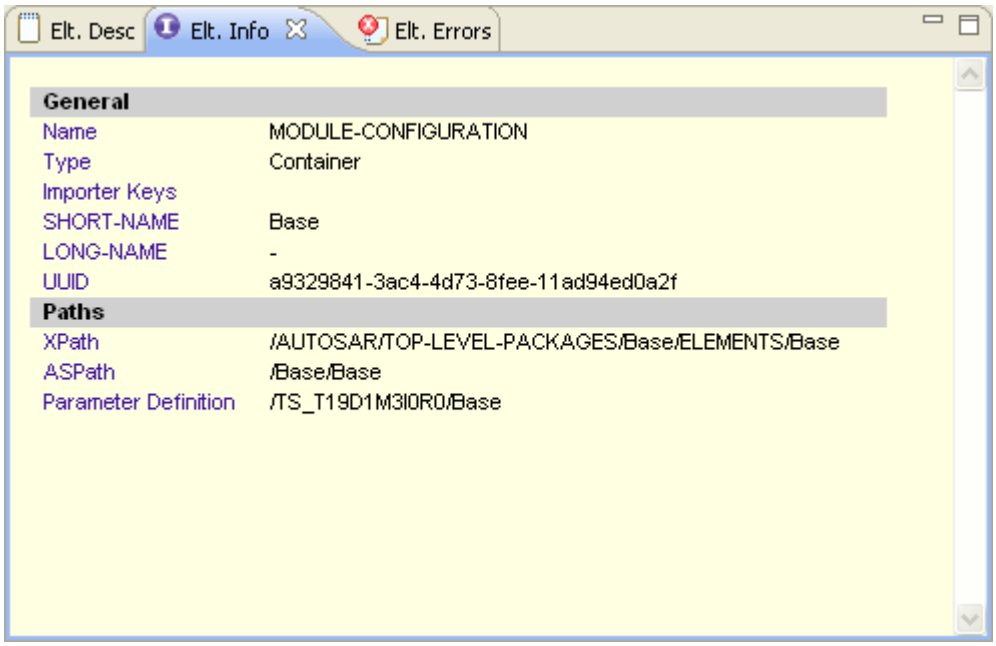
  The *Message Details* window opens up.



The window displays the following information:

- *Timestamp*: the time when the error was reported
- *Severity*: the message is an error or a warning
- *Message*: the message information

4.4.4.2.2. Element Info view



The *Element Info* view shows parameter meta information of the parameter currently seelcted in the editor. This meta information is derived from the Autosar parameter definition.

This may contain the following information:

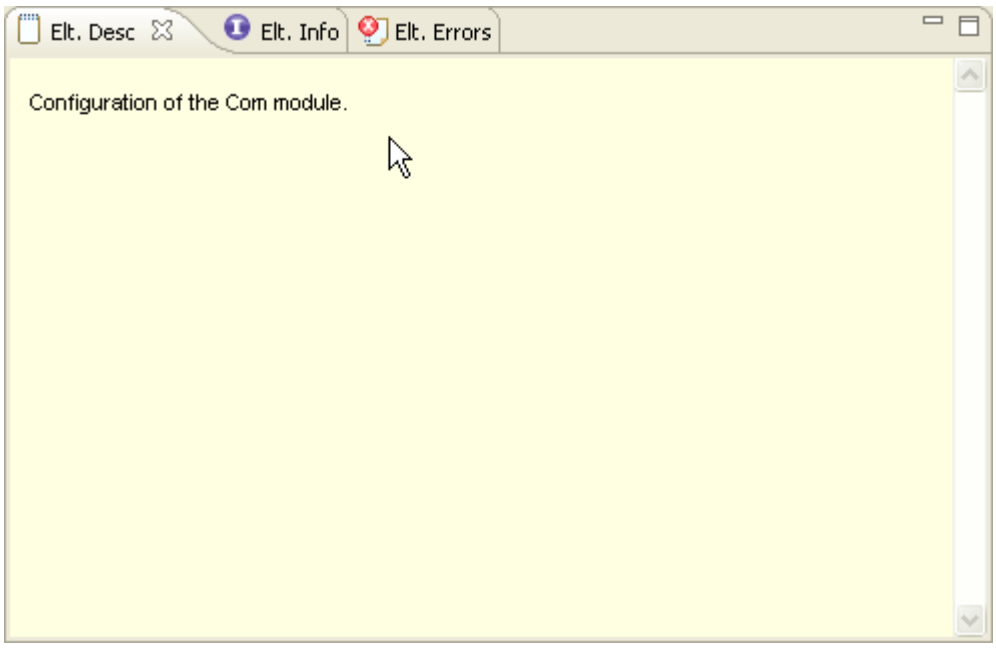
- *Name*: the name of the element

- *Type*: the element type (integer, floating point unit, Boolean, string, etc.)
- *Source*: where the source of the parameter (e.g. *manually edited* or *default value*).
- *SHORT-NAME*: the Autosar SHORT-NAME of the parameter
- *Min*: the minimum number of entries in a table
- *Max*: the maximum number of entries in a table
- *ORIGIN*: the Autosar ORIGIN of the parameter
- *UUID*: the Autosar UUID of the parameter
- *XPath*: the XPath path expression via which it is possible to navigate to the parameter
- *ASPath*: The Autosar path expression of the parameter. For reference parameters, the path describing all referenceable nodes is shown here. This path is also shown, if there is no referenceable node available (e.g. not yet configured). It may contain a prefix such as "ASPathDataOfSchema:" what means, that all configured nodes for a specific schema-node are referenceable. In this case, the Autosar SHORT-NAME-path of the schema-node is the last part of this path
- *Parameter Definition*: displays the path to the Autosar parameter definition element

Note

Not all information is available for every configuration parameter. E.g. the *ASPath* is omitted if the parameter is not referenceable by an Autosar path.

4.4.4.2.3. Element Description view



The *Element Description* view describes the parameter selected in the editor and gives configuration hints, references other parameters, etc. The description may contain hyperlinks to related parameters (not shown in the screenshot).

To navigate to the related parameter:

- Select the linked parameter from the *Element Description* view.

  This opens up information about the parameter in the *Element Description* view.

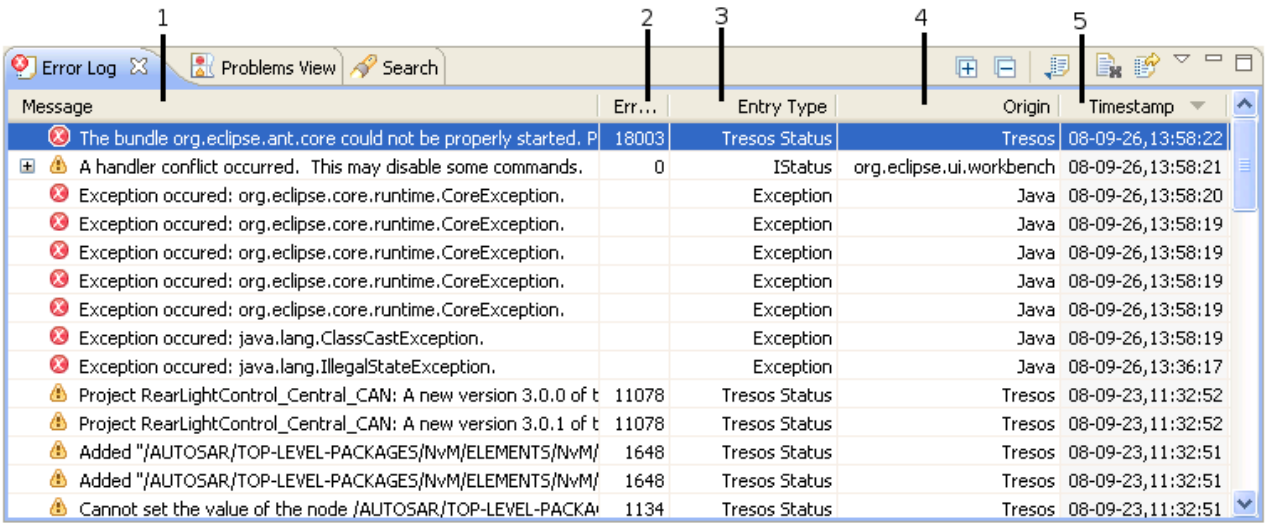
4.4.5. Error Log view

The *Error Log* view is where all logged errors are visible in the GUI. The *Error Log* receives error messages from EB tresos Studio components and from other framework components. By default the list of messages in the *Error Log* is preserved in between the sessions of EB tresos Studio, thus the *Error Log* may start up filled with messages.

Tip

Section Section 4.5.3.4: Setting Error Log preferences describes ways to customize the behaviour of the *Error Log*.

The default location of the *Error Log* view is on bottom of the screen in the middle.



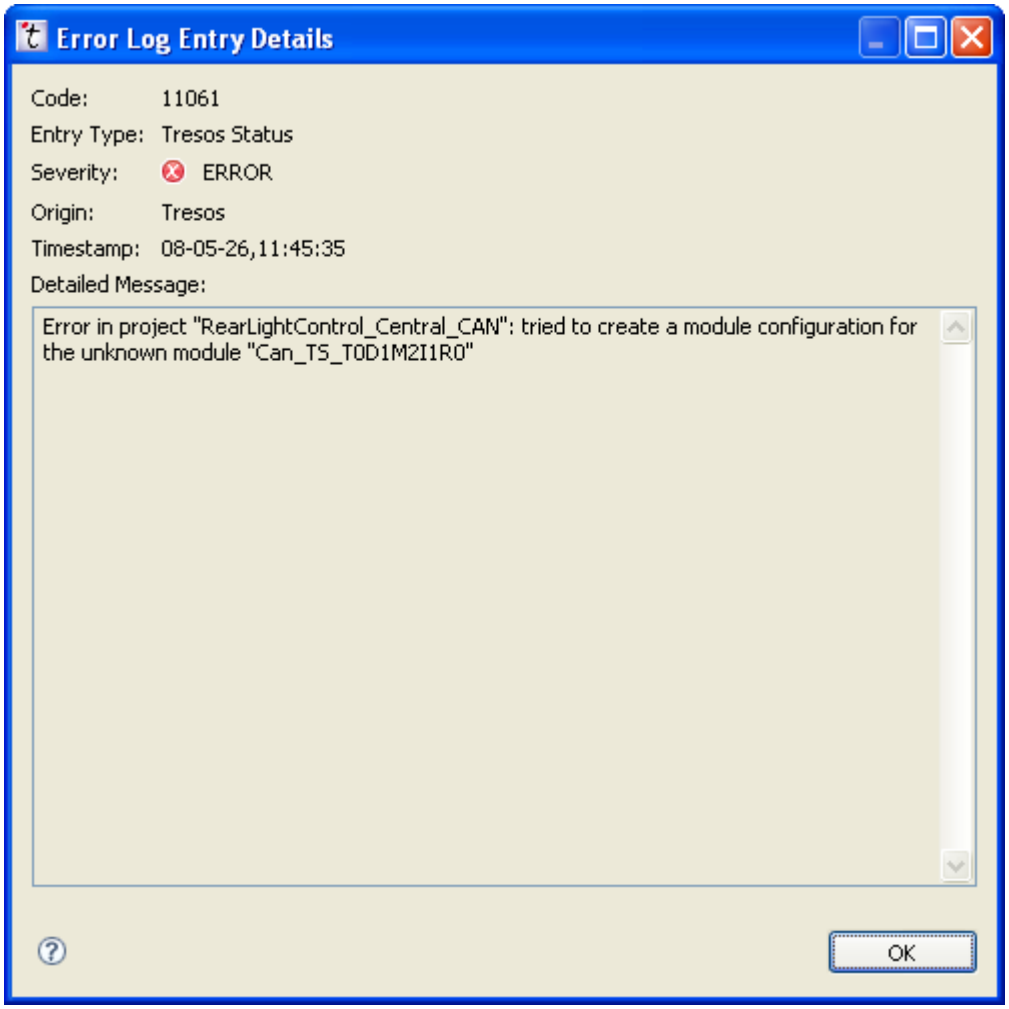
The *Error Log* displays a list of log entries, which can have the following properties:

Number	Information type	Explanation
1	Message	Indicates the actual error message. The type of message and its severity is indicated by the icon in front of the message. ► indicates an error message (something did not go as expected and it had to be aborted) ► indicates a warning message (something did not go as expected, but it could continue) ► indicates an info message (just a notification or feedback for the user)
2	Error Code	Indicates the error number, which is used by EB tresos Studio developers to identify the error in the software code.
3	Entry Type	Indicates the error type, e.g. an exception, the EB tresos status, etc.
4	Origin	Indicates the component that logged the message.
5	Timestamp	Indicates the day and time the message was produced.

To find out more detailed information about an error:

► Double-click on the entry you want to learn more about in the Error Log.

The *Error Log Entry Details* window opens up.



Additionally to the *Error Log* view information, the window displays a detailed message text.

4.4.5.1. Changing the display settings of the Error Log view

To display the message hierarchy of all log entries into messages and submessages:

- Select the icon.

  The view of all messages is expanded and all messages with submessages are displayed.

To show just the top-level messages instead of all submessages:

- Select the icon.

  The message view collapses and displays just top-level messages.

To write a message to a text file:

- Select the icon.

  A dialog window opens up, which enables you to save the message to your computer. This may be useful, e.g. if you want to contact EB tresos Studio support with a problem connected with the message.

To clear the Error Log view:

- Select the icon.

  All messages disappear from the Error Log view. The messages are not lost but can be restored with the restore button.

To restore messages to the Error Log view:

- Select the icon.

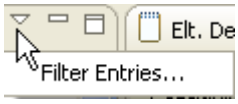
  All messages you previously cleared are restored to the Error Log view.

4.4.5.2. Filtering Error Log messages by their severity level

You may filter out messages depending on their severity level.

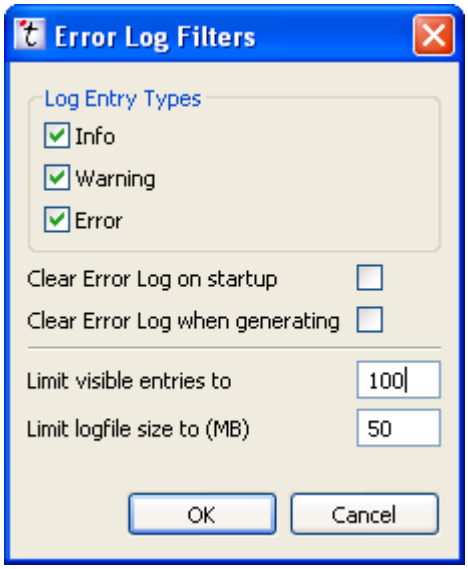
- Select the icon.

  A context menu opens up.



- Select *Filter Entries*.

  The *Error Log Filters* window opens up.



- Select the **Log Entry Types** you want the Error Log view to display.
- Select **OK**.

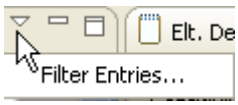
  The Error Log view displays only messages that you selected in the Error Log Filters window.

4.4.5.3. Restricting the amount of shown Error Log messages and the backup logfile size

You may restrict the amount of shown Error Log messages and the size of the backup logfiles.

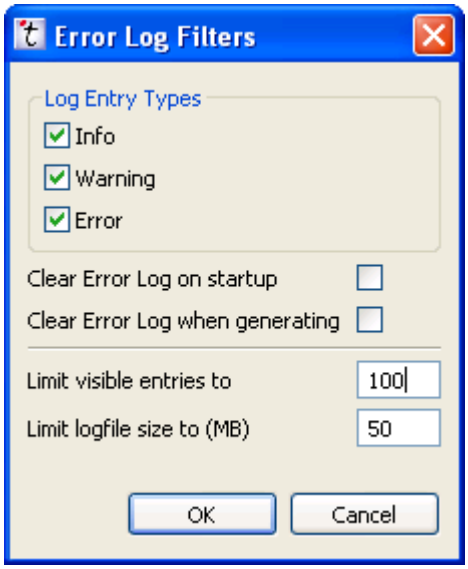
- Select the icon.

  A context menu opens up.



- Select *Filter Entries*.

  The *Error Log Filters* window opens up.



- Enter the amount of Error Log Entries you want to see at most.
- Enter the maximum size for backup logfiles.
- Select **OK**.

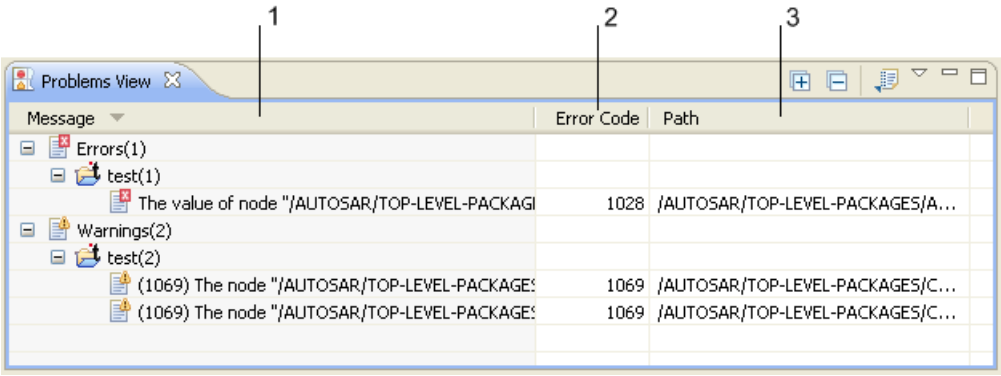
  The Error Log view displays at most the number of messages you configured. All backup logfiles created in the future will not be bigger than the configured value.

4.4.5.4. Copying Error Log entries into text documents

You can copy an Error Log entry as plain text to the Windows clipboard and paste it in any text application. This is useful e.g. if you want to contact EB support because of an error message.

To do so, right-click on the Error Log entry you want to copy and select **Copy**. A text representation (including more information than you can see in the GUI) of the Error Log entry is now inside the Windows clipboard and can be pasted into other applications.

4.4.6. Problems view



While you edit your configuration, EB tresos Studio constantly checks the configuration for configuration errors and warnings in the background. This background operation is called the *verifier* thread. If a problem in the configuration is found, the verifier adds an entry to the *Problems* view.

The items in the *Problems* view are grouped by the project names and by the type of problem. Possbile types are *error*, *warning* and *info*.

To display all levels of errors and warnings available:

- Open up the top-level tree of the *Problems* view.

  The number in brackets behind the group entries indicates the number of messages in the respective category.

Number	Information type	Explanation
1	Message	Indicates the message text. The message can be either a warning or an error.
2	Error Code	Indicates the error number, which is used by EB tresos Studio developers to identify the error in the software code.
3	Path	Indicates the XPath term in the configuration model to the parameter that issued a warning or error.

To edit the parameter that has issued a message:

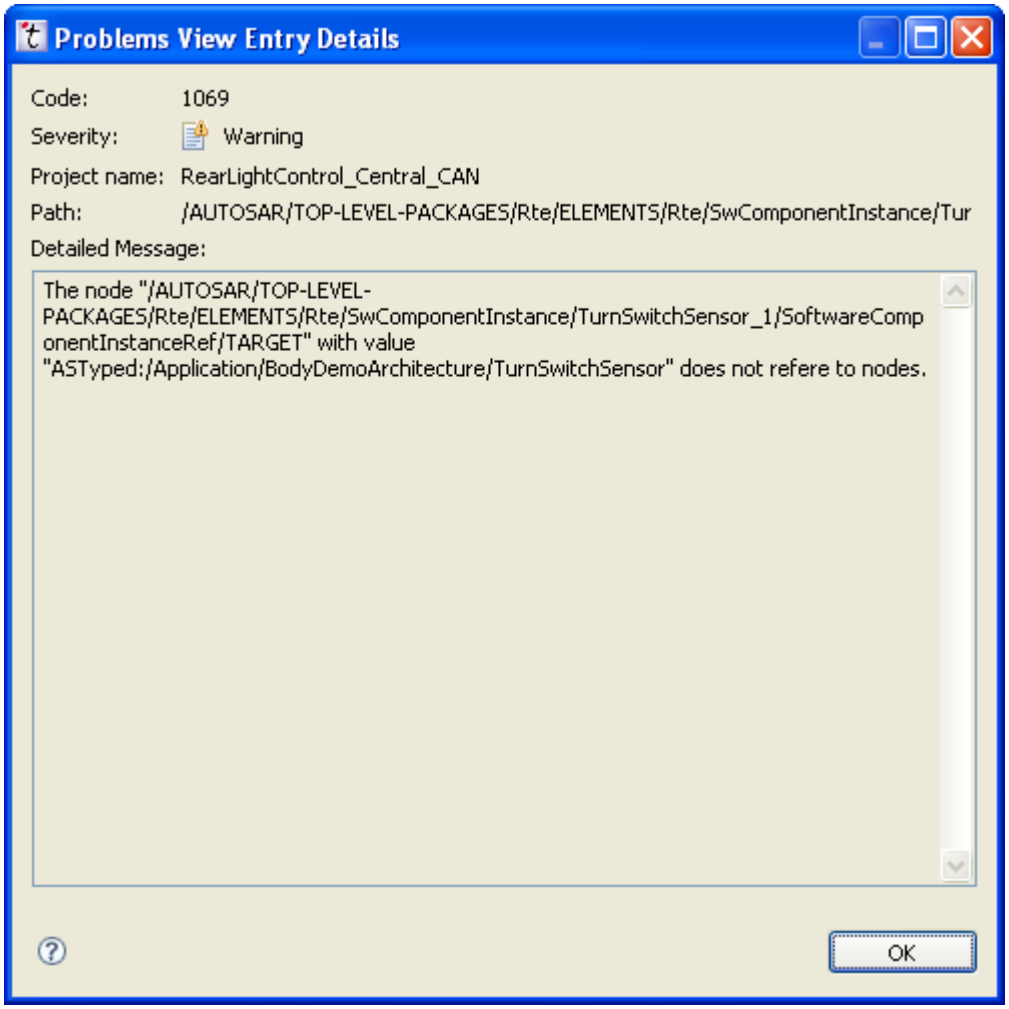
- Single-click on the entry in the *Problems* view.

  The *Editor* view of the respective parameter opens up directly at the position where the problem is and you may edit the parameter.

To find out more detailed information about the entry:

- Double-click on the entry in the *Problems* view.

  The Problems View Entry Details window opens up.



Additionally to the *Problems* view entry this window shows:

- the project name in which the message occurred
- the detailed message text

To display the message hierarchy of all log entries into messages and submessages:

- Select the icon.

  The view of all messages is expanded and all messages with submessages are displayed.

To show just the top-level messages instead of all submessages:

- Select the icon.

  The message view collapses and displays just top-level messages.

To write a message to a text file:

- Select the icon.

  A dialog window opens up, which enables you to save the message to your computer. This may be useful, e.g. if you want to contact EB tresos Studio support with a problem connected with the message.

4.4.6.1. Filtering Problems view messages by their severity level

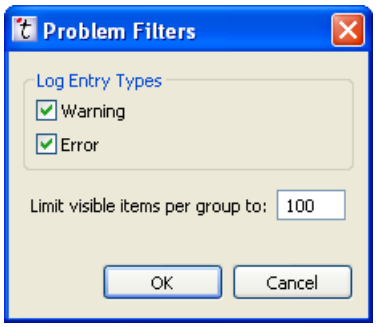
You may filter out messages depending on their severity level.

- Select the icon.

  A context menu opens up.

- Select *Filter Entries*.

  The *Problem Filters* window opens up.



- Select the *Log Entry Types* you want the *Problems* view to display.
- Select **OK**.

  The *Problems* view displays only the types that you selected in the *Problem Filters* window.

4.4.6.2. Limit the number of entries shown in the Problems View

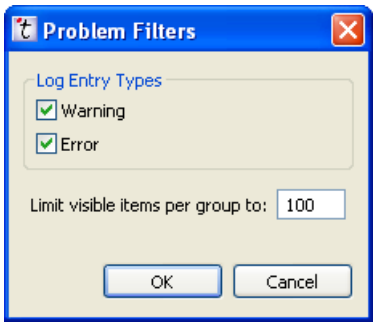
If you have a many entries in the *Problems* view and experience the GUI slowing down, you can restrict the number of entries shown in the *Problems* to an upper bound:

- Select the icon.

  A context menu opens up.

- Select *Filter Entries*.

  The *Problem Filters* window opens up.



- Enter a positive number in the entry **Limit visisble items per group to**.
- Select **OK**.

  The *Problems* view restricts the number of shown entries.

4.5. Changing the EB tresos Studio setup

This section describes how you can change several aspects of the EB tresos Studio setup, including installing new licenses, finding information about installed modules and plugins and changing the program-wide preferences.

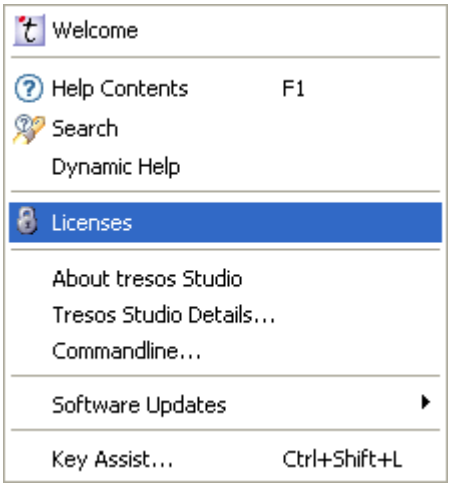
4.5.1. Managing licenses

EB tresos Studio provides a dialog window which lists all licenses currently installed and lets you remove or add licenses. You can also use this dialog to view information about each license like the licenseed features or the expiry date and you can verify if the license works properly.

To open the *License* dialog:

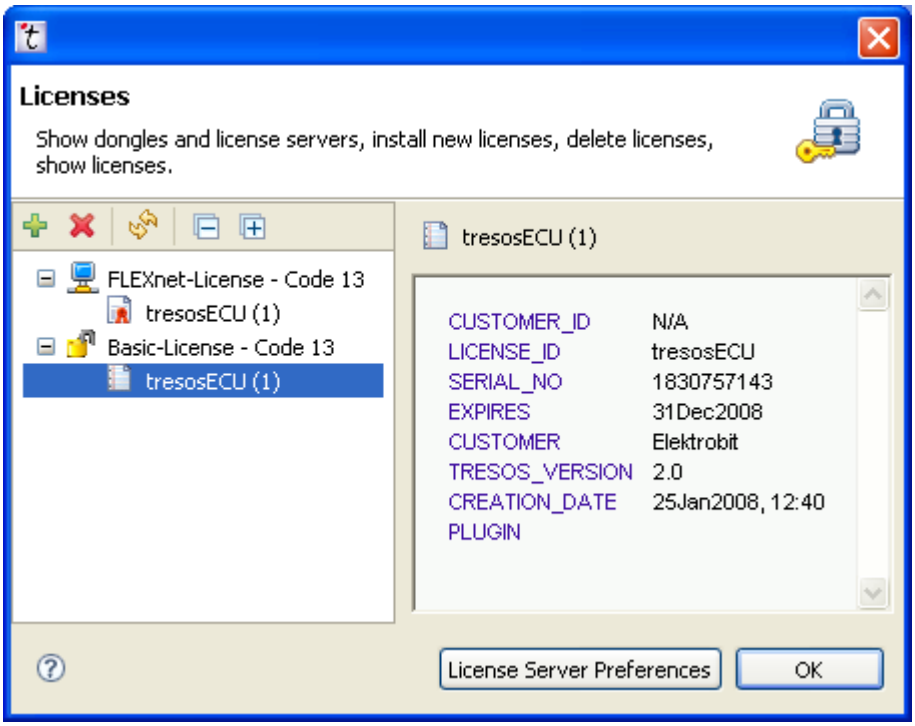
- Select the **Help** menu.

  A submenu opens up.



- Select *Licenses* from the submenu.

  The *Licenses* dialog opens up.



The dialog is split into two parts: a list of licenses on the left and an area where detailed information of a lincense is displayed. To view the detailed information you need to select one of the license on the left.

The list on the left is grouped by license type and customer ID. Possible license types are *FLEXnet License* (in case of a floating license), *Wibu-License* (in case of a dongled license) and *Basic-License* (in case of an undongled license). A *floating* license means that the license is verified by contacting a server in your computer network. A *dongled* license means that the license is verified by communitacting with a hardware device plugged into your computer's USB port (see the picture below). It is possible to have more than one license of one type. In this case, there are multiple items displayed in one group.

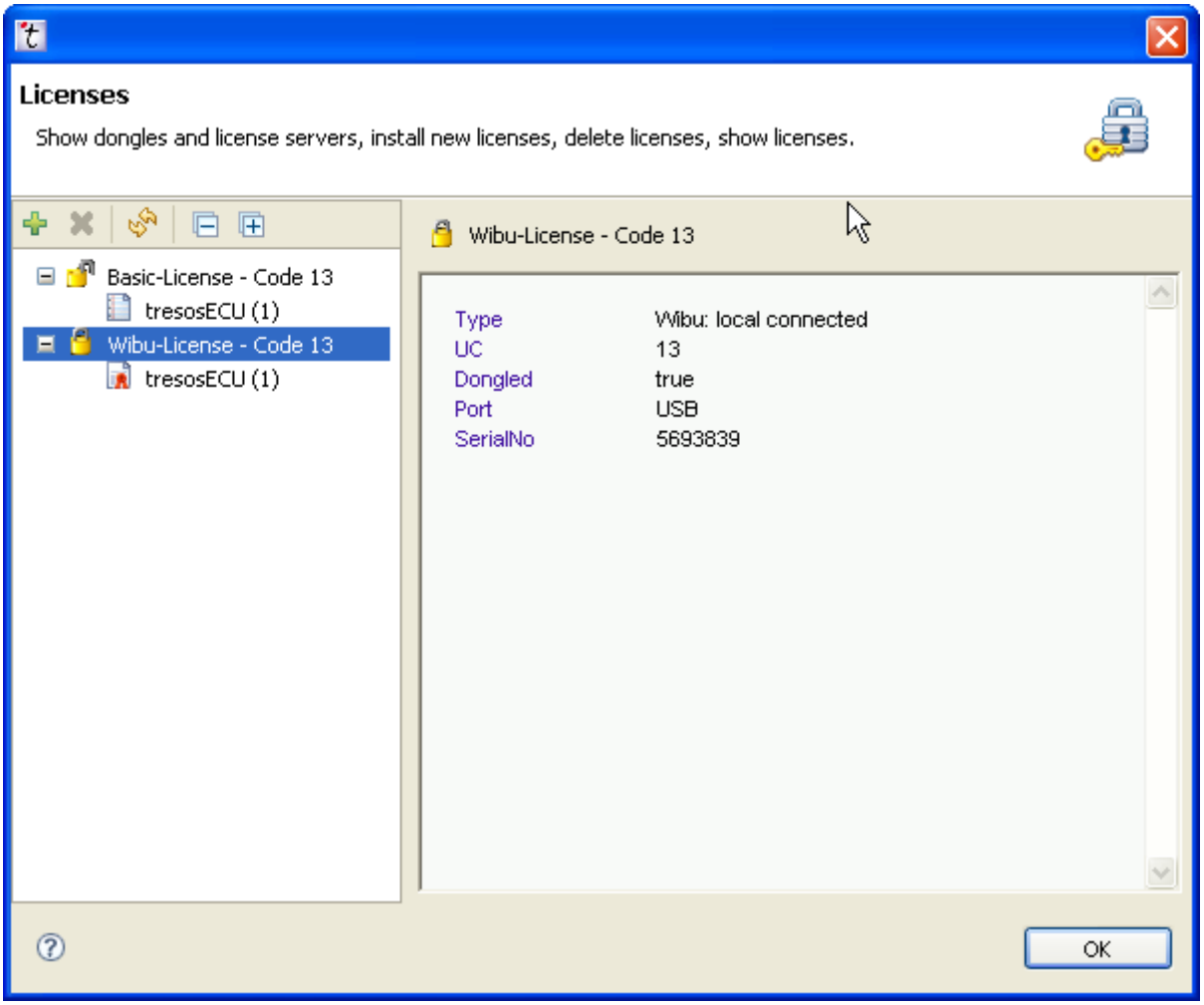
- Select a license from the list.

  The license details are displayed on the right side of the *Licenses* window.

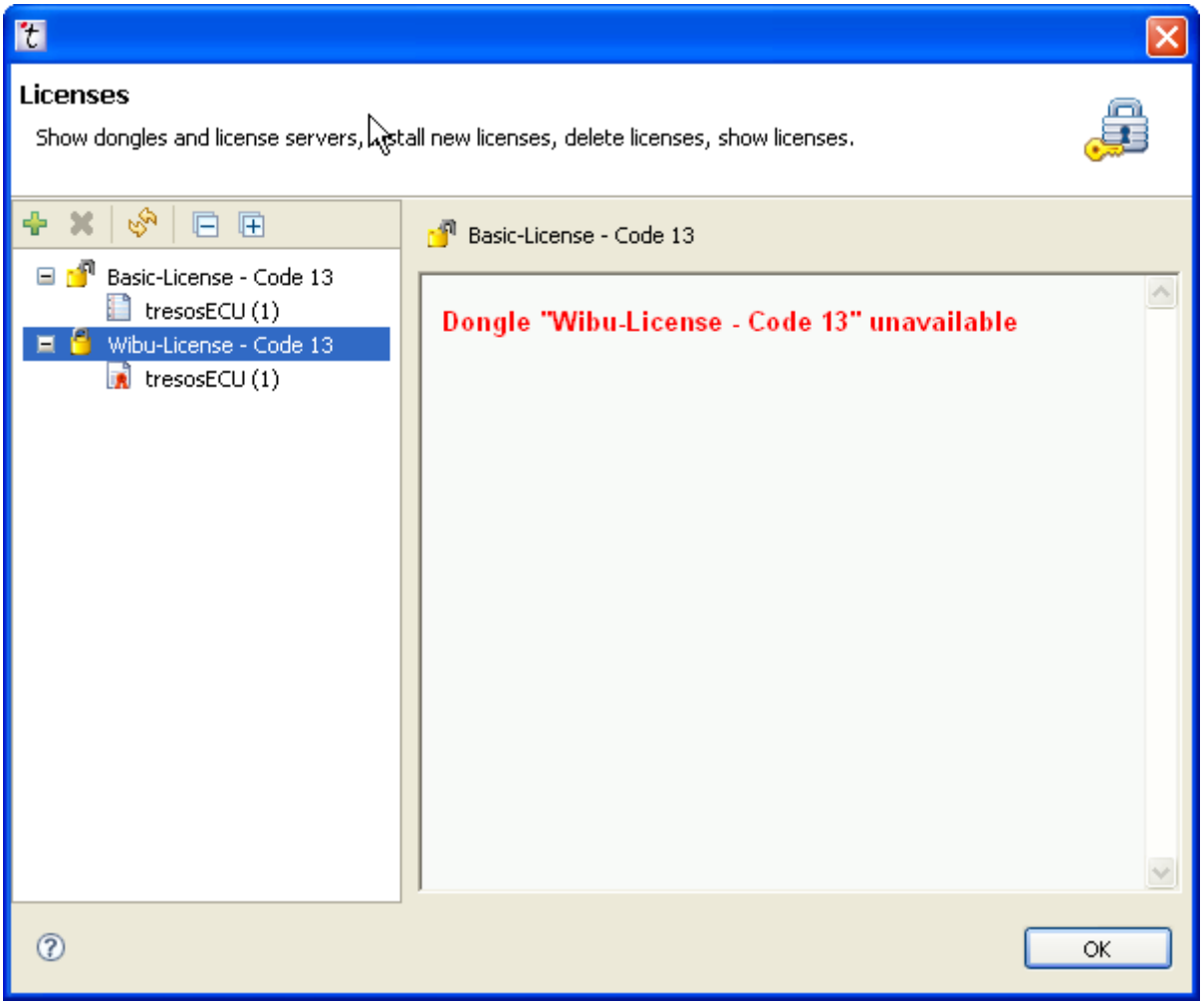
- The the icon expands the license list completely. Use this if you have a lot of licenses installed and want to see them all.

- The icon collapses the license list, so that only the groups are visible. Use this if you have a lot of groups expanded and want to revert the display to its initial state.

- If you use Floating Network licenses, press the button **License Server Preferences** to configure the License Server. See Section 4.5.3.5: Setting License Server preferences for details.



Example 4.1. The example shows information about a WibuKey dongle.



Example 4.2. The example shows a dongle that cannot be accessed by EB tresos Studio.

4.5.1.1. Adding a license

If you have purchased a new license, you have to add it to EB tresos Studio before you can use it. To add a license:

- Select the green cross icon .

  A file dialog window opens up.
- Browse your computer for licenses in the *.lic* format.
- Select the license.
- Select **Open**.

  The license is added to your list in the *Licenses* window.

4.5.1.2. Deleting a license

To delete a license:

- Select the license from the list of the *Licenses* window.
- Select the icon.

  This deletes the selected license from your list.

4.5.1.3. Refreshing license information

To refresh license information:

- Select the icon with two arrows to update system information.

  Your computer installs all available licenses from the configuration directory and updates the information whether and which licenses and dongles are available.

4.5.2. Viewing setup information

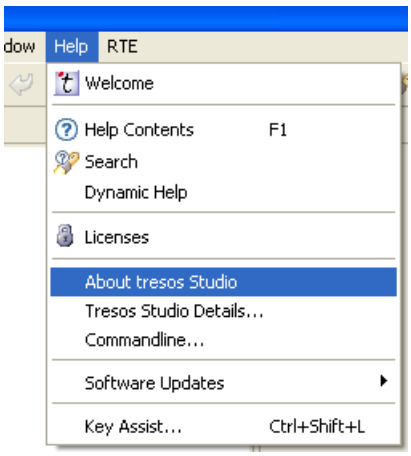
EB tresos Studio provides two dialogs which display information about the program and its setup: the *About* dialog and the *Tresos Details* dialog. This section will sow you how these dialogs can be opended and will describe the information they contain.

4.5.2.1. Finding the EB tresos Studio version, the build number, and the available features

The EB tresos Studio version, the build number and the available features can be viewed in the *About* dialog. To open the *About* dialog:

- Select the **Help** menu.

  A context menu opens up.



► Select *About tresos Studio* from the context menu.

The *About* dialog opens up.

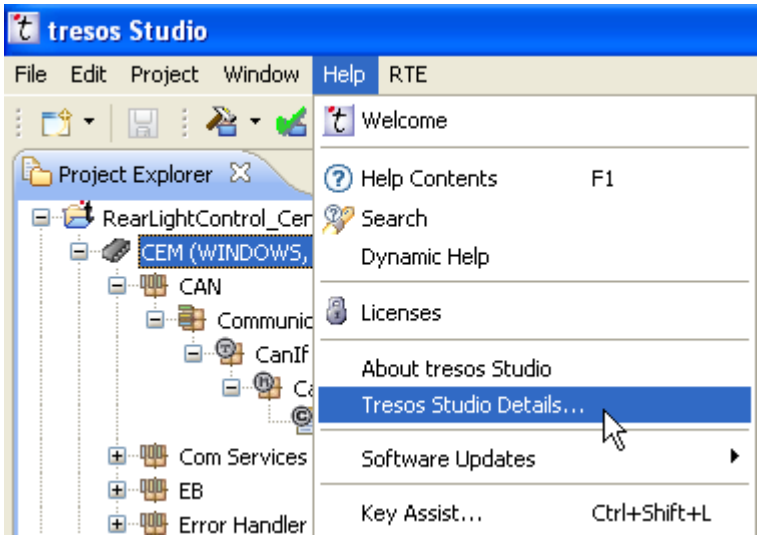
Directly in the center of the dialog you can see the EB tresos Studio version you are currently using, e.g. *2008.b*. In the line starting with *Build* the build number is shown. You can use this number when you are contacting EB support to identify the exact version you are using. At the line starting with *Features* the supported versions of the Autosar standard are listed.

4.5.2.2. Finding detailed setup information via the *Tresos Details* dialog

To open the dialog window for viewing detailed setup information:

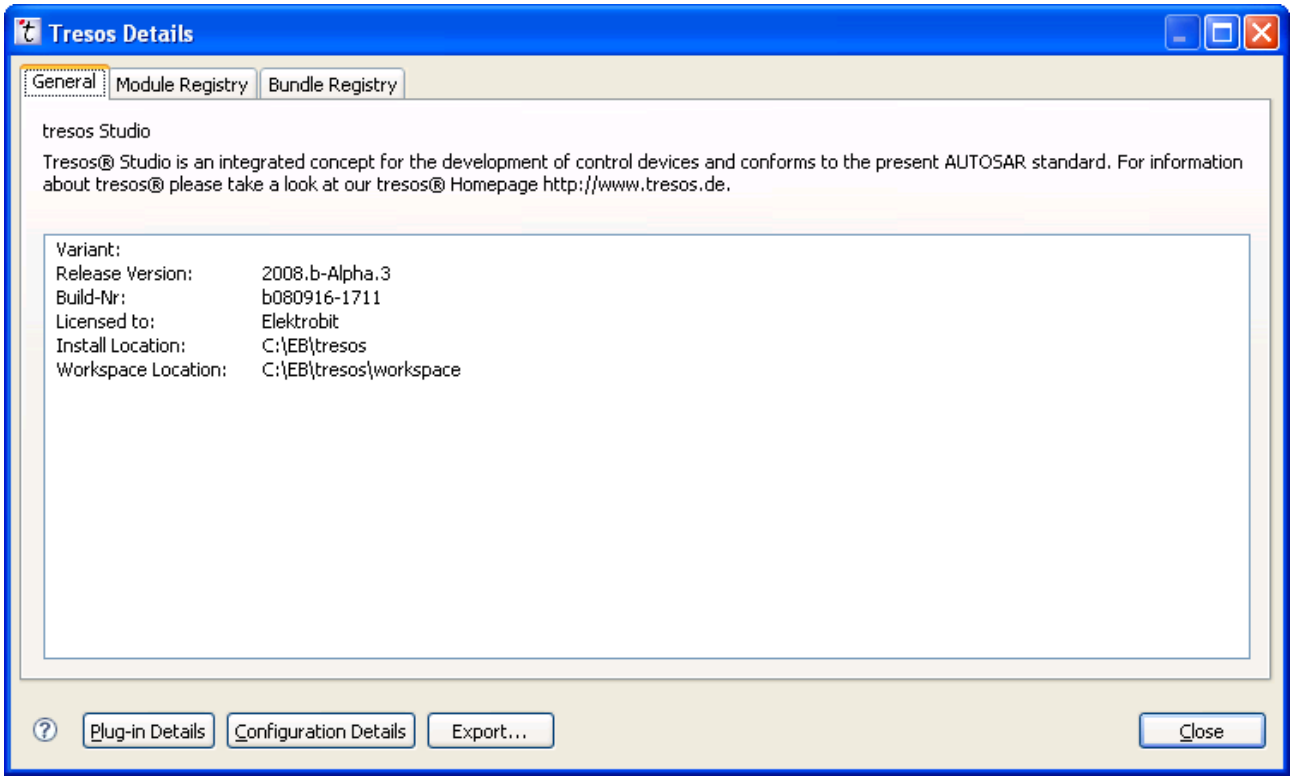
- Select the **Help** menu.

  A context menu opens up.



- Select *Tresos Studio Details* from the context menu.

  The *Tresos Details* window opens up.



4.5.2.3. Finding the location of the installation and the workspace

Select the *General* tab to display:

- the build number,
- the name of the licensee,
- the installation location on your computer, and
- the workspace location on your computer.

4.5.2.4. Finding all installed modules

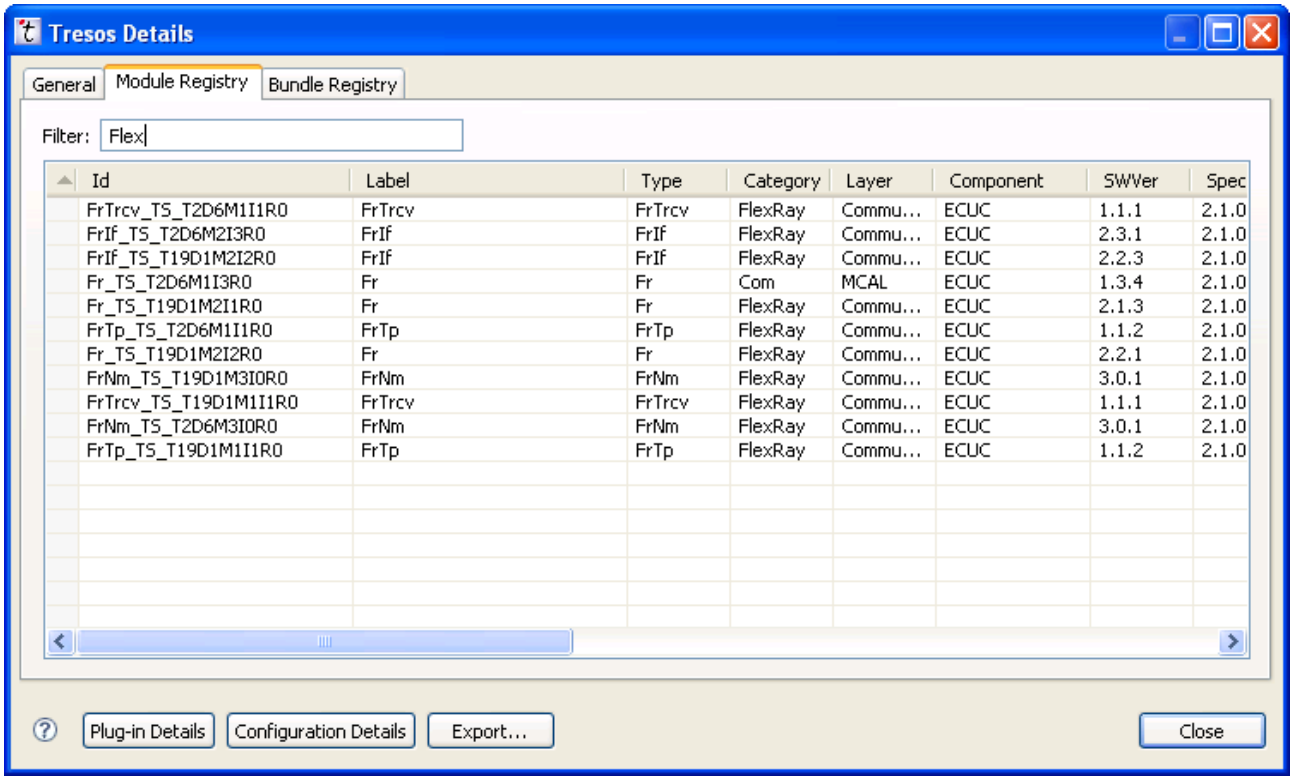
To find out if a specific module is part of your installation and which Autosar specification version it implements:

- Select the Module Registry tab.

  The list below the tab displays all installed modules.

- Type the text you are looking for into the *Filter* textbox.

  The table in the *Module Registry* tab now shows just entries that contain text that matches your entered text. To view all entries again, just clear the text control by deleting its content.



Id	Label	Type	Category	Layer	Component	SWVer	Spec
FrTrcv_TS_T2D6M1I1R0	FrTrcv	FrTrcv	FlexRay	Commu...	ECUC	1.1.1	2.1.0
FrIf_TS_T2D6M2I3R0	FrIf	FrIf	FlexRay	Commu...	ECUC	2.3.1	2.1.0
FrIf_TS_T19D1M2I2R0	FrIf	FrIf	FlexRay	Commu...	ECUC	2.2.3	2.1.0
Fr_TS_T2D6M1I3R0	Fr	Fr	Com	MCAL	ECUC	1.3.4	2.1.0
Fr_TS_T19D1M2I1R0	Fr	Fr	FlexRay	Commu...	ECUC	2.1.3	2.1.0
FrTp_TS_T2D6M1I1R0	FrTp	FrTp	FlexRay	Commu...	ECUC	1.1.2	2.1.0
Fr_TS_T19D1M2I2R0	Fr	Fr	FlexRay	Commu...	ECUC	2.2.1	2.1.0
FrNm_TS_T19D1M3I0R0	FrNm	FrNm	FlexRay	Commu...	ECUC	3.0.1	2.1.0
FrTrcv_TS_T19D1M1I1R0	FrTrcv	FrTrcv	FlexRay	Commu...	ECUC	1.1.1	2.1.0
FrNm_TS_T2D6M3I0R0	FrNm	FrNm	FlexRay	Commu...	ECUC	3.0.1	2.1.0
FrTp_TS_T19D1M1I1R0	FrTp	FrTp	FlexRay	Commu...	ECUC	1.1.2	2.1.0

4.5.2.5. Finding all installed plug-ins

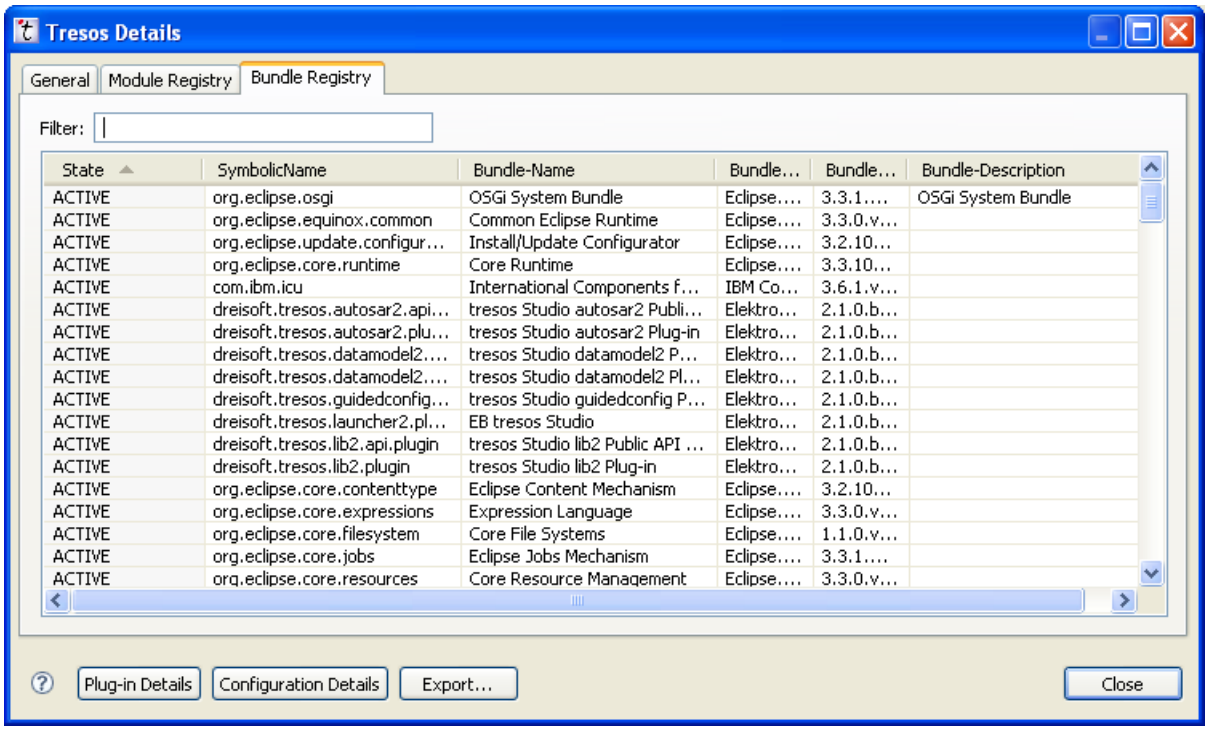
To find out if a specific plug-in is part of your EB tresos Studio installation and if it is working properly:

- Select the *Bundle Registry* tab.

  The list below the tab displays all installed plug-ins.

- Type the text you are looking for into the *Filter* text control.

  The table in the *Bundle Registry* tab now shows just entries that contain text that matches your entered text. To view all entries again, just clear the text control by deleting its content.



State	SymbolicName	Bundle-Name	Bundle...	Bundle...	Bundle-Description
ACTIVE	org.eclipse.osgi	OSGi System Bundle	Eclipse....	3.3.1....	OSGi System Bundle
ACTIVE	org.eclipse.equinox.common	Common Eclipse Runtime	Eclipse....	3.3.0.v...	
ACTIVE	org.eclipse.update.configur...	Install/Update Configurator	Eclipse....	3.2.10...	
ACTIVE	org.eclipse.core.runtime	Core Runtime	Eclipse....	3.3.10...	
ACTIVE	com.ibm.icu	International Components f...	IBM Co...	3.6.1.v...	
ACTIVE	dreisoft.tresos.autosar2.api...	tresos Studio autosar2 Publi...	Elektro...	2.1.0.b...	
ACTIVE	dreisoft.tresos.autosar2.plu...	tresos Studio autosar2 Plug-in	Elektro...	2.1.0.b...	
ACTIVE	dreisoft.tresos.datamodel2....	tresos Studio datamodel2 P...	Elektro...	2.1.0.b...	
ACTIVE	dreisoft.tresos.datamodel2....	tresos Studio datamodel2 Pl...	Elektro...	2.1.0.b...	
ACTIVE	dreisoft.tresos.guidedconfig...	tresos Studio guidedconfig P...	Elektro...	2.1.0.b...	
ACTIVE	dreisoft.tresos.launcher2.pl...	EB tresos Studio	Elektro...	2.1.0.b...	
ACTIVE	dreisoft.tresos.lib2.api.plugin	tresos Studio lib2 Public API ...	Elektro...	2.1.0.b...	
ACTIVE	dreisoft.tresos.lib2.plugin	tresos Studio lib2 Plug-in	Elektro...	2.1.0.b...	
ACTIVE	org.eclipse.core.contenttype	Eclipse Content Mechanism	Eclipse....	3.2.10...	
ACTIVE	org.eclipse.core.expressions	Expression Language	Eclipse....	3.3.0.v...	
ACTIVE	org.eclipse.core.filesystem	Core File Systems	Eclipse....	1.1.0.v...	
ACTIVE	org.eclipse.core.jobs	Eclipse Jobs Mechanism	Eclipse....	3.3.1....	
ACTIVE	org.eclipse.core.resources	Core Resource Management	Eclipse....	3.3.0.v...	

Note

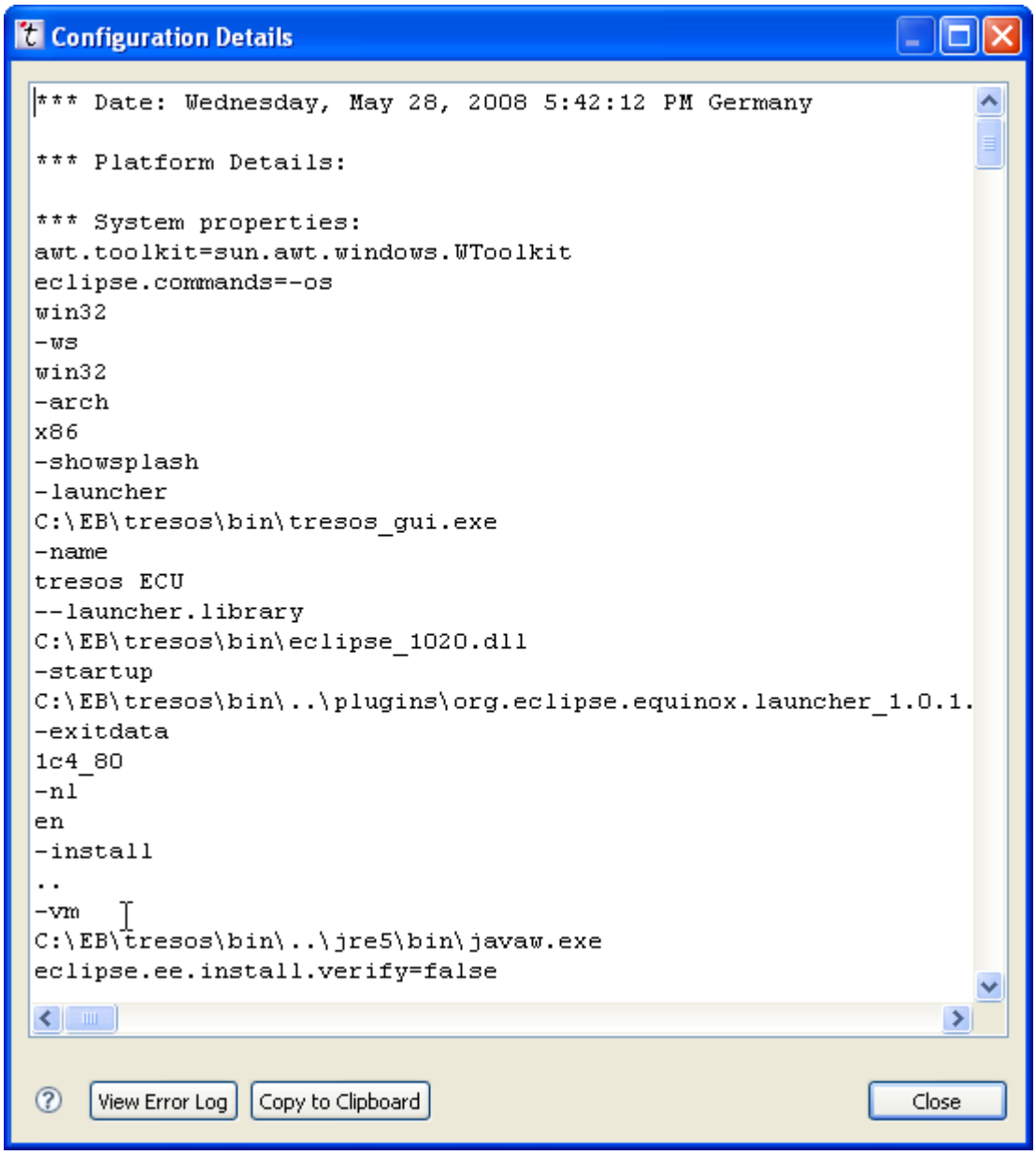
This list shows all plugins from the Eclipse framework upon which EB tresos Studio is built as well as EB tresos Studio core libraries. If a plugin could not be loaded properly, the line is colored red.

4.5.2.6. Finding program startup parameters and system property settings

To view internal information about how EB tresos Studio was launched and what internal system properties are set:

- In the *Tresos Details* window, select the **Configuration Details** button.

The *Configuration Details* window opens up.



This window provides detailed configuration information about program startup parameters and system property settings. In case of problems this information can be useful for EB support.

4.5.2.7. Exporting a system report of your EB tresos Studio installation

If you are having problems with your EB tresos Studio installation, it is useful for EB support to have as much information about your setup as possible. You can help EB support by attaching a system report of your EB tresos Studio installation to any support ticket. To generate this report as a text file:

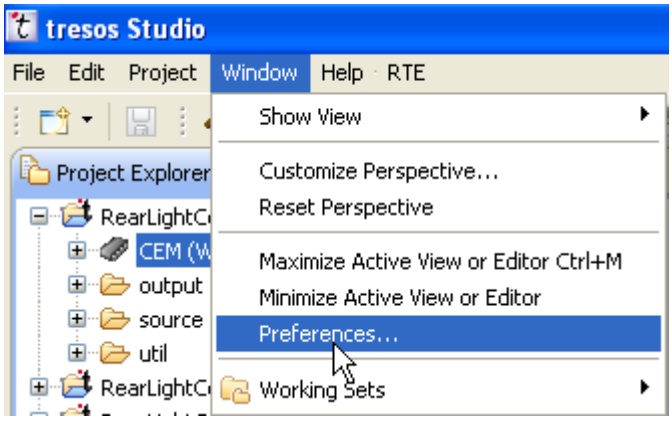
- In the *Tresos Details* window, select the **Export** button.

  This opens up a dialog window, with which you may save the data as file to your computer.

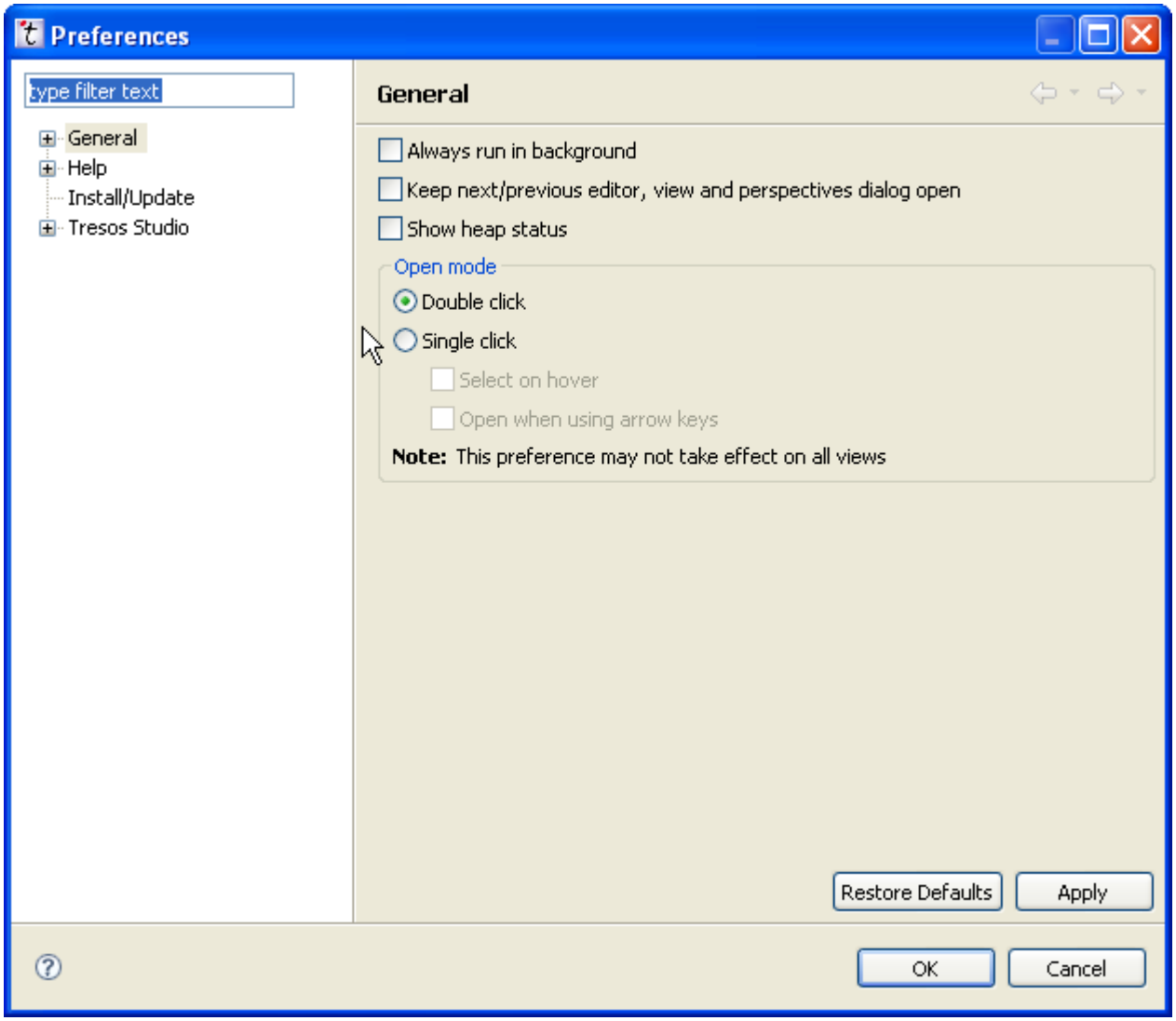
4.5.3. Setting preferences

To globally store settings for a workspace set your preferences in the Window menu.

- Select **Window**.
- Select *Preferences* from the drop down menu.

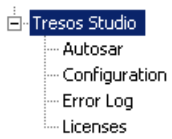


The *Preferences* window opens up.



- Select *Tresos Studio* from the tree view on the left of the window.
- Double-click *Tresos Studio*.

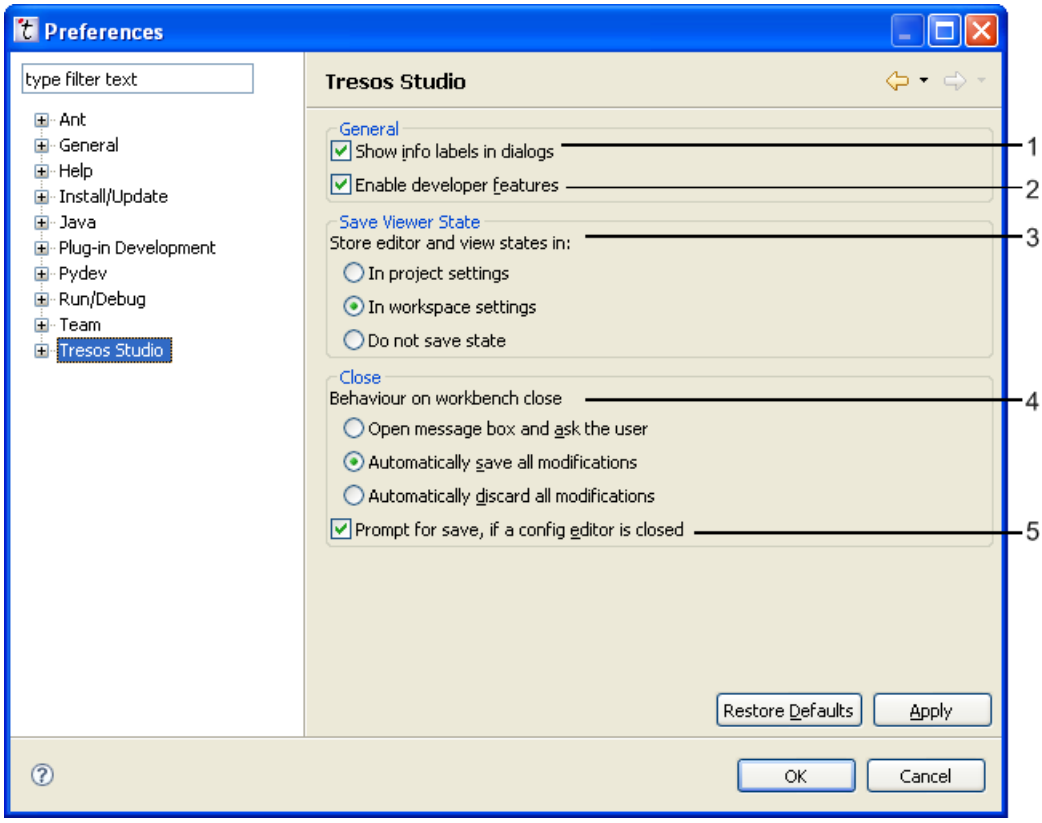
  A subtree opens up underneath the tree item *Tresos Studio*.



The following subsection will explain the preference pages related to items of the subtree. On any page, you can press the *Restore Defaults* button to restore the default values of the page. You can press the **Apply** button to store the changes you made so far. To save the changes and close the complete dialog, press the **Ok** button. To cancel any changes you made and to close the dialog, press the **Cancel** button.

4.5.3.1. Setting general preferences

To set preferences about the general behaviour of EB tresos Studio, use the following settings on the *Preferences Tresos Studio* window:



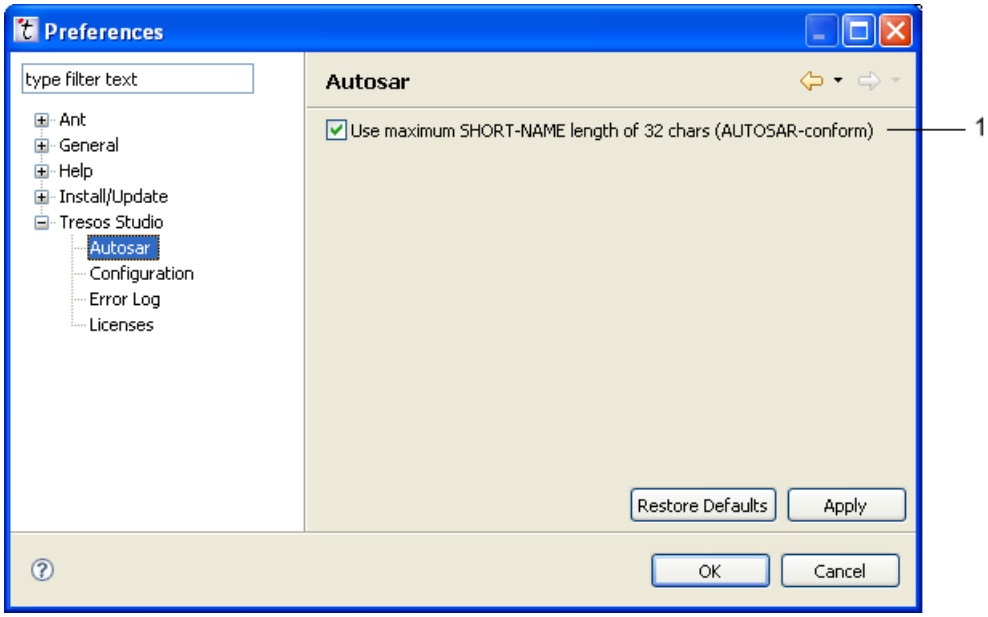
Number	Checkbox/Radiobutton	Instruction
1	**Show info labels in dialogs**	Select to enable help text for dialogs and wizards.
2	**Enable developer features**	Enables some features that are relevant for developers, who develop their own Autosar Basic Software modules. This is described in a more detailed way in the Developer's user's guide, see Chapter 5: EB tresos Studio developer's guide
3	**Store editor and view states in:**	The configuration editors of EB tresos Studio are able to preserve their current state, e.g. columns shown in tables, column width, number format selected between editing sessions, by storing the state to disk when the editor is closed. The **Store editor and view states in** allows to configure where this information is saved to.

Number	Checkbox/Radiobutton	Instruction
		In project settings The editor state is saved inside the EB tresos Studio project the editor belongs to. The state is written to files in a directory `.settings` beneath the project directory of the project. **In workspace settings** The state is saved into the meta-data area of the workspace currently opened. The meta-data area is stored in a hidden directory called `.metadata` beneath the workspace root directory. **Do not save state** The editor does not write or load any state.
4	**Behaviour on workbench close**	**Open message box and ask the user** Select to be asked every time you close the workbench whether you want to save changes. This will prevent you from closing the program with unsaved projects. **Automatically save all modification** Select if you want to save changes to your projects automatically upon closing. No dialog window will appear. **Automatically discard all modifications** Select if you do not want to save changes to your projects automatically and do not want to be reminded to save by a dialog window.
5	**Prompt for save…**	Select if you want the configuration editor to remind you to save your settings before closing.

4.5.3.2. Setting Autosar preferences

To set Autosar preferences:

- select the Preferences Tresos Studio subpage *Autosar* and use the following settings:



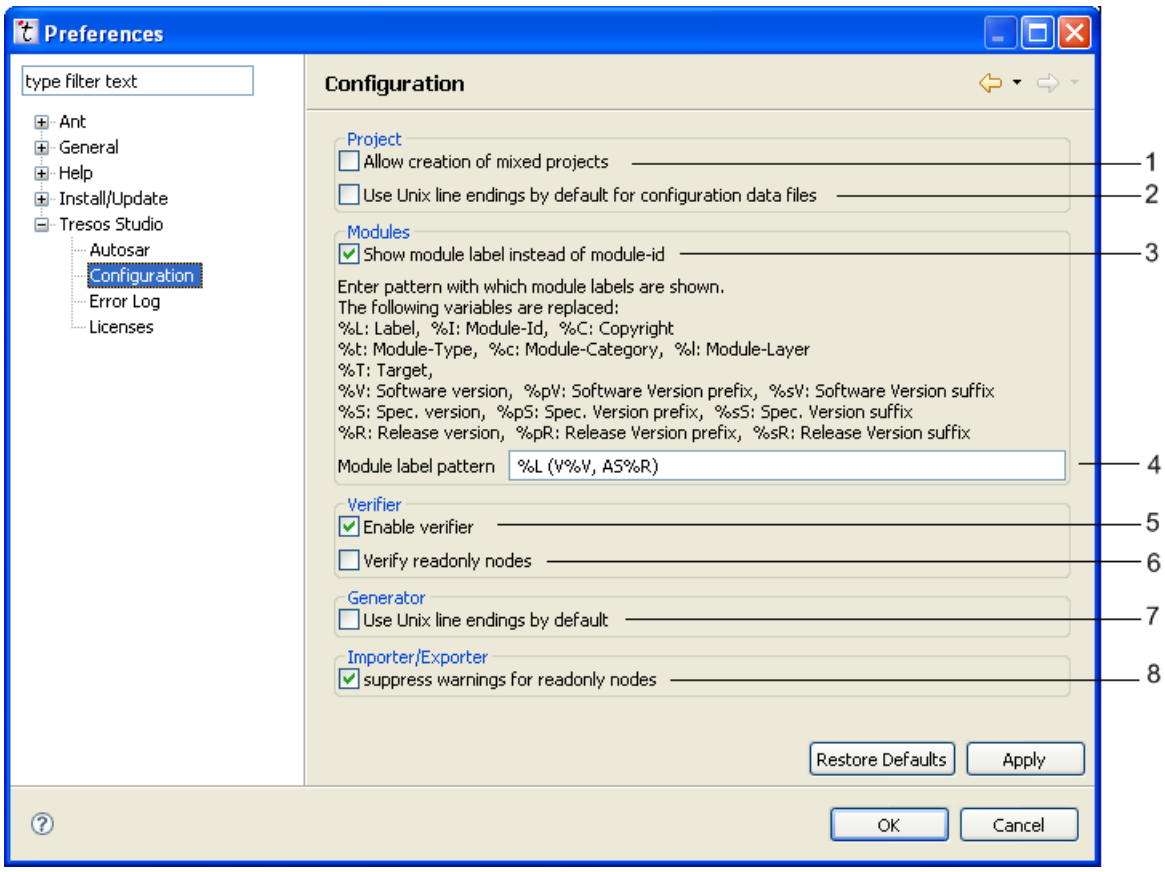
- Select **Use maximum SHORT-NAME length of 32 chars (AUTOSAR-conform)** to restrict the length of SHORT-NAME's to 32 chars according to the Autosar specification.

  If unchecked, longer names are allowed within tresos Studio. To export such a configuration to an Autosar file format, the validation must be deactivated. Problems may occur if other tools try to import this configuration.

4.5.3.3. Setting configuration preferences

To set configuration preferences:

- select the Preferences Tresos Studio subpage *Configuration* and use the following settings:



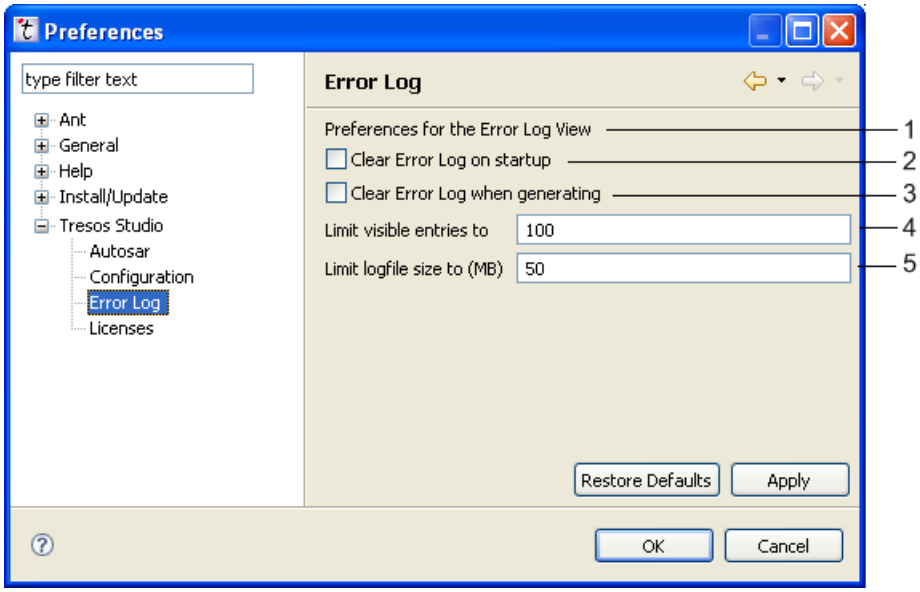
Number	Label	Instruction
1	**Allow creation of mixed projects**	Select to configure modules with different release versions. A mixed project may contain module configurations with different release versions. If you enable this feature, the *Release Version* combobox in the project wizard (see Section 4.6.1: Creating a new project) contains the entry *mixed*.
2	**Use Unix line endings by default for configuration files**	Select to use Unix line endings instead of platform line endings in configuration data files. This is a project-specific configuration. You may change it for each project in the preferences. The configuration data files are located inside the config directory of the project directory. The Exporters also use this setting.
3	**Show module label instead ...**	If this checkbox is checked, the GUI uses the pattern defined in (4) to display a module. This setting is visible e.g. in the *Project Browser* or the *Add Module* dialog.
4	**Module label pattern**	Enter the pattern defining how the module label should look in the GUI.

Number	Label	Instruction
5	**Enable verifier**	Select to enable/disable the verifier. **Note** The verifier is a thread that constantly runs in the background to check if the current configuration of all loaded projects is valid. The verifier creates entries that are displayed in the Problems view **Warning** No code will be generated, if you disable the verifier while there are still errors in the Problems view.
6	**Verify read only nodes**	Select to verify the read-only nodes (nodes that cannot be changed).
7	**Use Unix line endings by default**	Select to use Unix line endings in your generated code as default. You may override this workspace-wide default setting in the project preferences for every project individually. See chapter `Viewing and changing project properties` Section 4.6.7: Viewing and changing project properties for details.
8	**Suppress warnings for readonly nodes**	Select if you want to supress warnings for readonly nodes while running importer or exporter.

4.5.3.4. Setting Error Log preferences

To set Error Log preferences:

- Select the *Preferences Tresos Studio* subpage *Error Log*

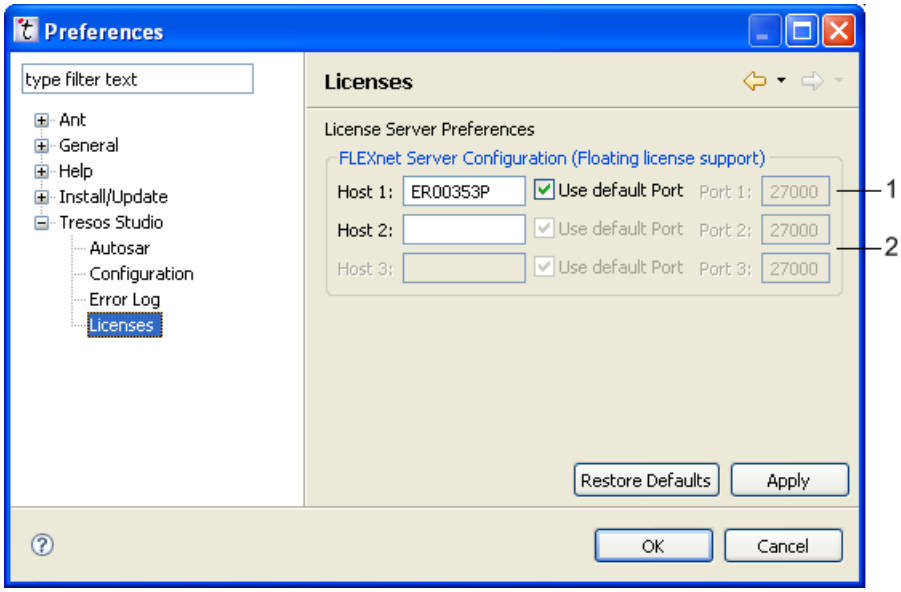


- Select **Clear Error Log** on startup, if you want the *Error Log* view to contain no messages when you start EB tresos Studio.
- Select **Clear Error Log** button, if you want to remove all messages from the error log. You can do this e.g. before generating code, to clearly see which messages are logged during code generation.
- To limit the number of shown entries in the *Error Log* enter the respective number of entries you wish to be displayed in the **Limit visible entries to** combobox. Entering big numbers may slow down the GUI.
- All errors are written to a logfile called `.tresoslog`. With increasing size of the logfile, the GUI may become slower and the memory usage might increase. The parameter **Limit logfile size to (MB)** provides an upper bound after which the content of the log is discarded and written to a backup file.

4.5.3.5. Setting License Server preferences

To configure the hostname and port for the FlexLM license server:

- Select the *Preferences Tresos Studio* subpage *Licenses*



Here you can define up to three flexlm license servers hostnames and their ports. This is a three-server redundancy configuration, but usually it is sufficient to only configure one server. It is recommended to use the default-port (automatically checks serveral ports).

To provide a default configuration which can be loaded by clicking on "Restore Defaults", a file with the name "flexlm.cfg" must be located within the "bin"-directory of your tresos-installation containing the following information:

```
# Contains the three-server redundancy configuration flexlm
# floating licensing server(s) and port(s). Each line must
# contain the hostname and an optional port. It is recom-
# mended to suppress the port (what means, that the default
# port shall be used
localhost
SomeHost:0815
127.0.0.1
```

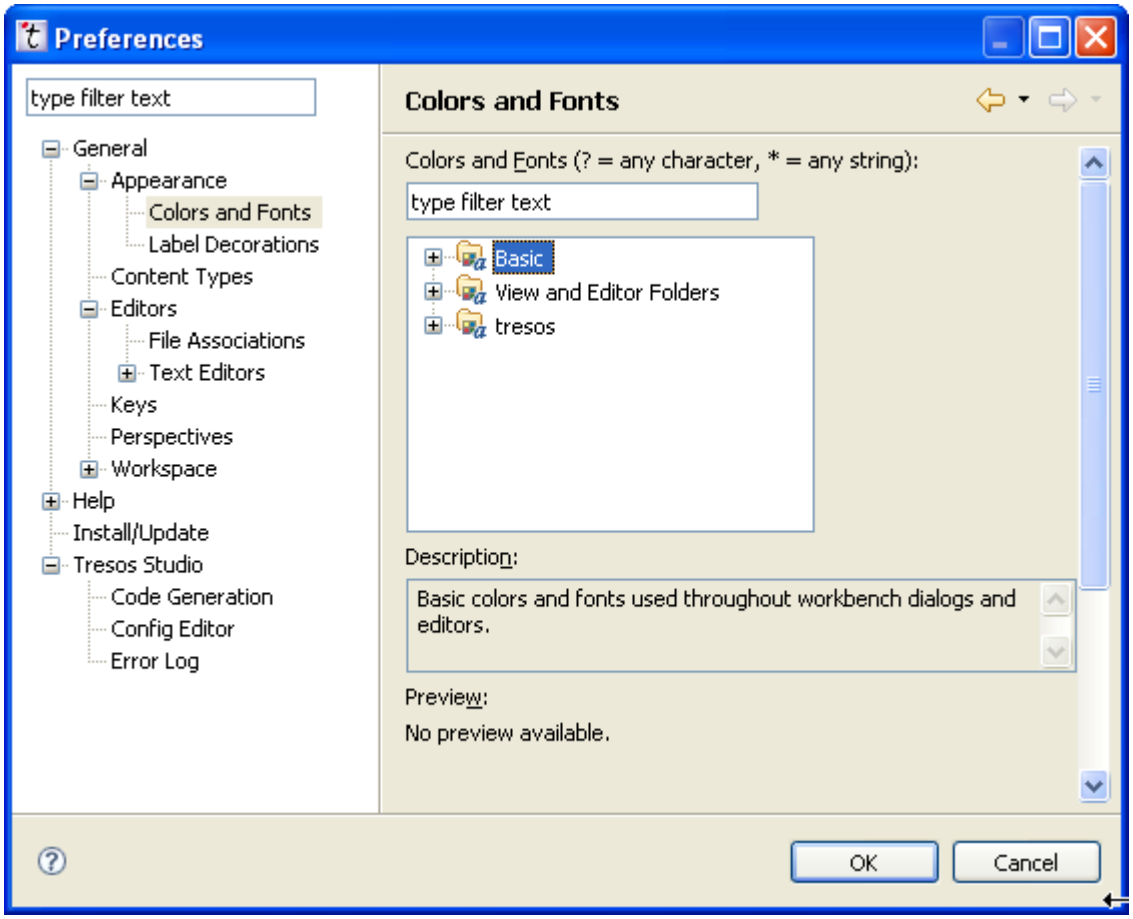
4.5.3.6. Setting colors and fonts

To set colors and fonts:

- Open the *Preferences* window and select *General* tree view on the left side.
- Open the tree view, select *Appearance*.
- Open the tree view under *Appearance* and select *Colors and Fonts*.

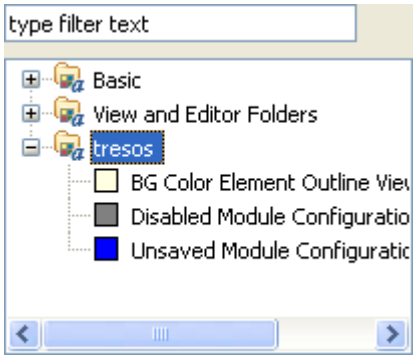
- Select the tresos folder in the *Colors and Fonts* view on the right hand side.

  You see a list of colors used in EB tresos Studio under **Basic** , **View and Editor Folders** , and **tresos**.



- Select **tresos**

  A subtree opens up.

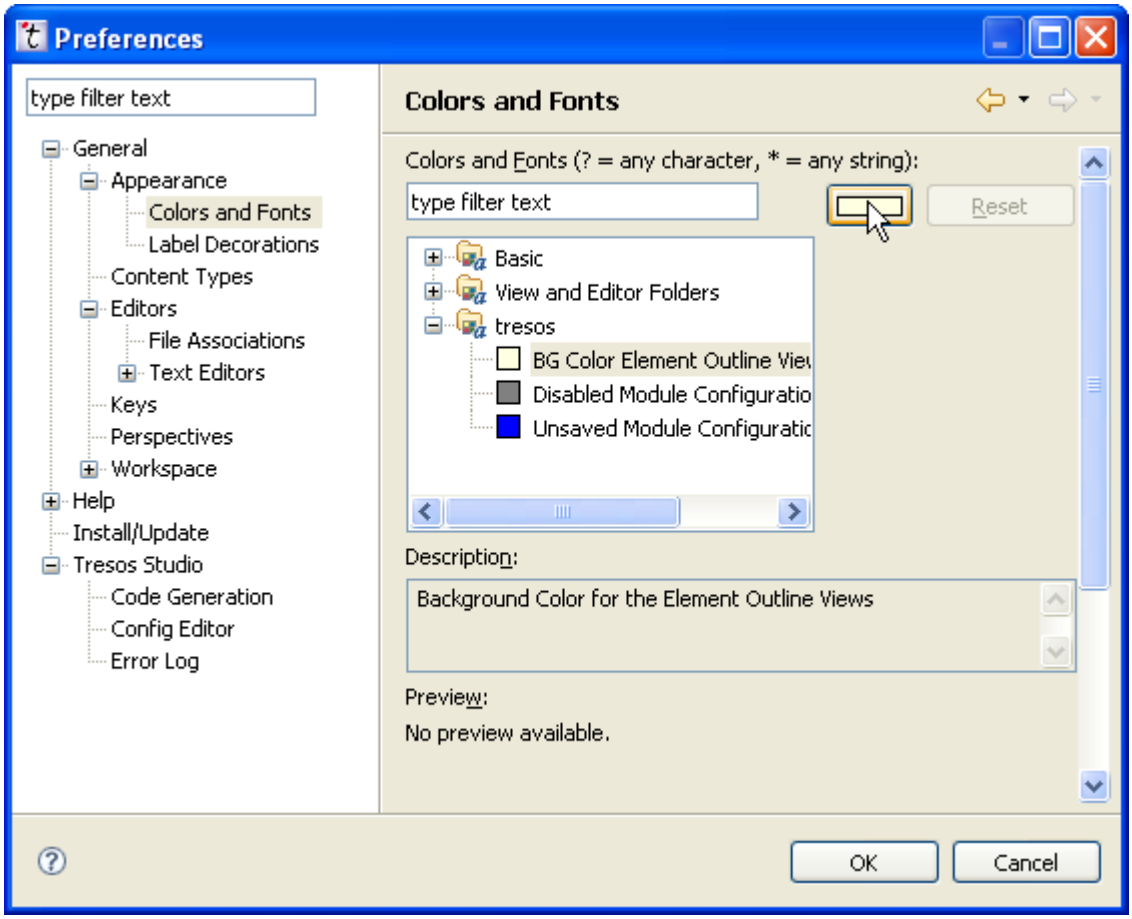


Checkbox	Explanation
BG Color Element Outline View	This item controls the background color of the*Element Description* and *Element Info* views.
Disabled Module Configuration (Foreground)	The default text color for modified and not yet saved module configurations is blue. This color is used in the*Project Explorer* and in the *Editor* view, for example.
Unsaved Module Configuration (Foreground)	The default text color for disabled module configurations is gray. This color is used in the *Project Explorer*, for example.

To change any of the colors:

- Select a color in the *Colors and Fonts* view on the right side.

  The color will appear on the right beside the **Reset** button.



- Select the color beside the **Reset** button.

  A window with color selections opens up.

- Select a color of your liking and click **OK**.

EB tresos Studio now displays the new color in the setting you chose.

4.5.4. Working with multiple workspaces to store different projects and settings

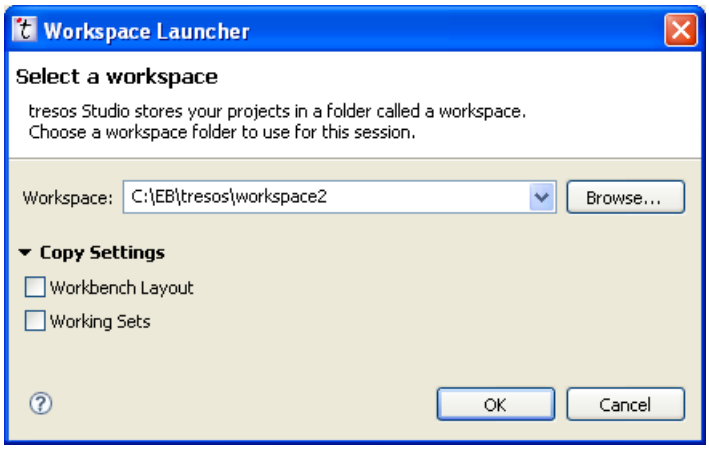
A *workspace* in EB tresos Studio contains all your projects and assoiciated information like your preferences (see Section 4.5.3: Setting preferences). If you want to keep some projects completely separated from others, or if you want to work with different preferences, you can use multiple workspace and switch to the workspace you are planning to use.

Note

The default workspace is located directly inside your EB tresos Studio installation directory and is called `workspace`. If you did not change anything you are using this workspace.

4.5.4.1. Switching the workspace

To switch the workspace, open the **File** menu and select *Switch workspace* and then *Other*. The *Workspace launcher* dialog opens:



Use the textbox labeled *Workspace* to type in your new workspace location. You can also use the **Browse** button to open up a Windows explorer to select the new location. If you have already used the switch workspace feature before, the directories used previously are available by clicking the drop-down arrow of the textbox.

If you click on *Copy settings* the dialog expands and you get more options (as depicted in the screenshot). Select **Workbench layout** if you have modified the view layout of EB tresos Studio and want to use the same layout in your new workspace. If you do not choose this option, the default layout will be used and EB tresos Studio looks like when you started it for the first time.

4.5.4.2. Finding out which workspace directory is used

If you want to check which workspace directory EB tresos Studio is currently using, open the *Tresos Details* dialog (see Section 4.5.2.3: Finding the location of the installation and the workspace). In the list on the *General* tab under *Workspace location* you can see the directory of the current workspace.

4.5.5. Customizing the main window

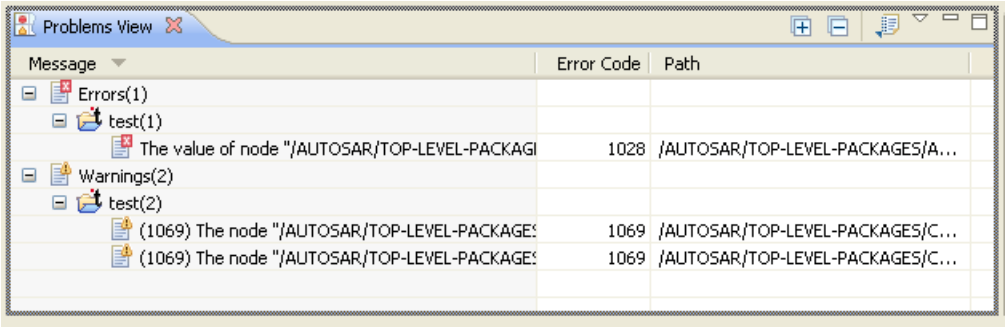
EB tresos Studio starts up with a default layout for the views. All the views you need during your work are present and they are located at the best place for small to medium sized screens. Sometimes you might want to change that layout so that it better suits your task. To do so you can rearrange the views via drag and drop; you can stack them; or you can maximize, minimize and close them.

Note

Some views are not visible by default, but they appear when they are needed. The *Search* view for example is usually not visible. Only if you have started a search the view appears and displays the search results to you.

4.5.5.1. Moving views to a different location

Every view has a title area. If you click this area, keep the mouse button pressend and move around you can drag the view to a different location. The location at which the view will be placed when you release the mouse button is indicated by a grey rectangle.

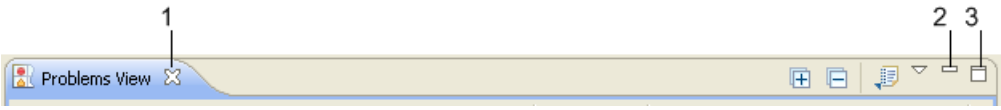


If you drop the view at a place where already another view is present, the views will be stacked. This means that there is always one view visible and the other view is behind it. The title areas of the views then vork like tabs and you can move the other view to the forground by clicking on its title area.

Tip

In the default window layout, the Element Outline views (see Section 4.4.4.2: Element Outline views) are an example for stacked views.

4.5.5.2. Minimizing, maximizing and closing views



Number	Function	Description
1	*Close*	Closes the view so that it disappears completely. If you want to reopen the view again, continue reading at Section 4.5.5.3: Opening a view.
2	*Minimize*	Minimizes the view. Minimized views occupy a small space at the border of the window. In this place only the icons of the minimized views are shown and you can click them to restore the views again. This is useful if you temporarily want to hide a view, e.g. to have more space for other views.
3	*Maximize*	Maximizes the view. The view will occupy the complete window. This is useful if you temporarily want to focus on one view only. E.g. sometimes you might want to maximize the *Problems* view in order to see the complete message text of all configuration problems.

4.5.5.3. Opening a view

Sometimes you might want to open a new or a closed view. To do so, open the **Window**, **Show View** menu choose *Other*. A dialog will open up which lets you choose all available views by their title and icon.

Note

There are many more views available than EB tresos Studio makes use of. In most cases you will find the view you are looking for in the *tresos Studio* section.

4.5.5.4. Resetting all views to the default layout

If you want to reset the views to their default layout, open the **Window** menu and choose *Reset perspective*. The arrangement of the views will look just like when you started EB tresos Studio for the first time.

4.5.6. Using EB tresos Studio with version control systems

If you follow a few guidelines, it is possible to use EB tresos Studio together with version control systems.

4.5.6.1. Placing a EB tresos Studio installation under version control systems

It is possible to place a complete EB tresos Studio installation under version control. You can submit the complete EB tresos Studio installation directory to your version control system, except the following files and directories:

- `workspace/`,
- `configuration/.settings/`, and
- `configuration/org.eclipse.*`.

Some version control systems mark files as read-only when you check them out. It this case you need to make all files from the EB tresos Studio installation directory writable by the current user.

4.5.6.2. Placing a projects under version control systems

It is possible to place projects under version control. To do so:

- Create a project (see Section 4.6.1: Creating a new project), but do not choose the default location. Instead, choose a directory outside of the `workspace` folder.
- Place that project directory under version control.
- On another computer, get a local revision of the project directory. If your version control system marks files as read-only when you check them out, you need to make the complete project directory writable by the current user.
- Use that EB tresos Studio installation on the second computer to import the project (see Section 4.11.1.1: Importing a project). Do not choose to copy the project into your workspace. This way, you are using the project directory which is under version control.

4.6. Working with projects

This section describes how to work with projects in EB tresos Studio. A project contains all the confgurations for one ECU. Before being able to edit module configurations, or to generate code, you need to create a project first and add modules to the project. The project also stores meta-information about the ECU like e.g. the ECU ID or the target and derivate setting.

4.6.1. Creating a new project

To create a new configuration project, EB tresos Studio provides the *Project Wizard*. The Project Wizard guides you through the creation of a new project via several dialog pages. You proceed from one page to the next by

pressing the **Next** button. You can always return to a previous page and change your settings by pressing the **Back** button. Only when you press the **Finish** button your settings are executed and a new project is created.

Note

Most of the wizard's buttons stay disabled until you have entered all required data.

4.6.1.1. Step 1: Starting the Project Wizard

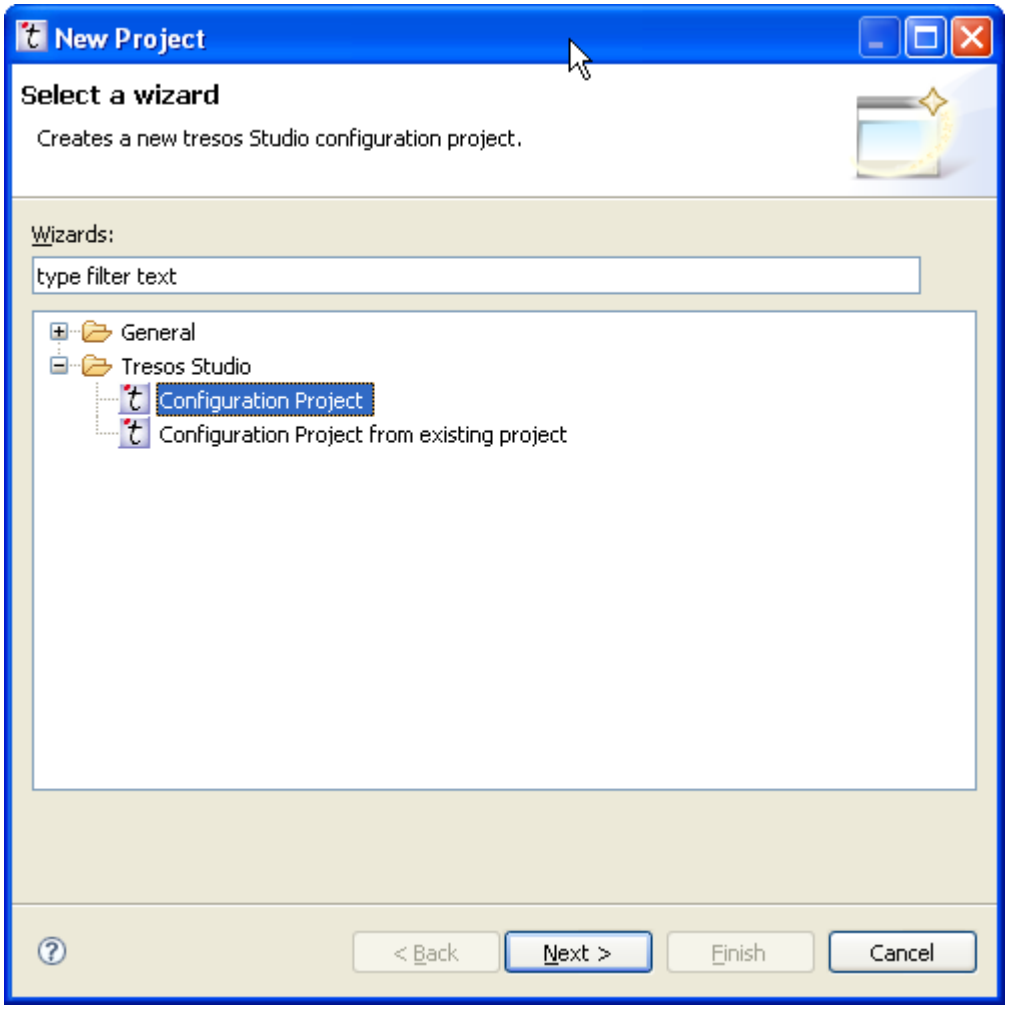
To start the Project Wizard, there are two alternatives:

- Alternative 1: Select a project node in the Project Explorer (see Section 4.4.3: Project Explorer view).
- Right-click on the node.

  A context menu opens up.
- Select *New*.

  A context menu opens up.
- Select *Project*.

  The Project Wizard *Select a wizard* window opens up.



Alternative 2: to start the Project Wizard,

- Select the **File** menu.

  A context menu opens up.

- Select *New*.

  A context menu opens up.

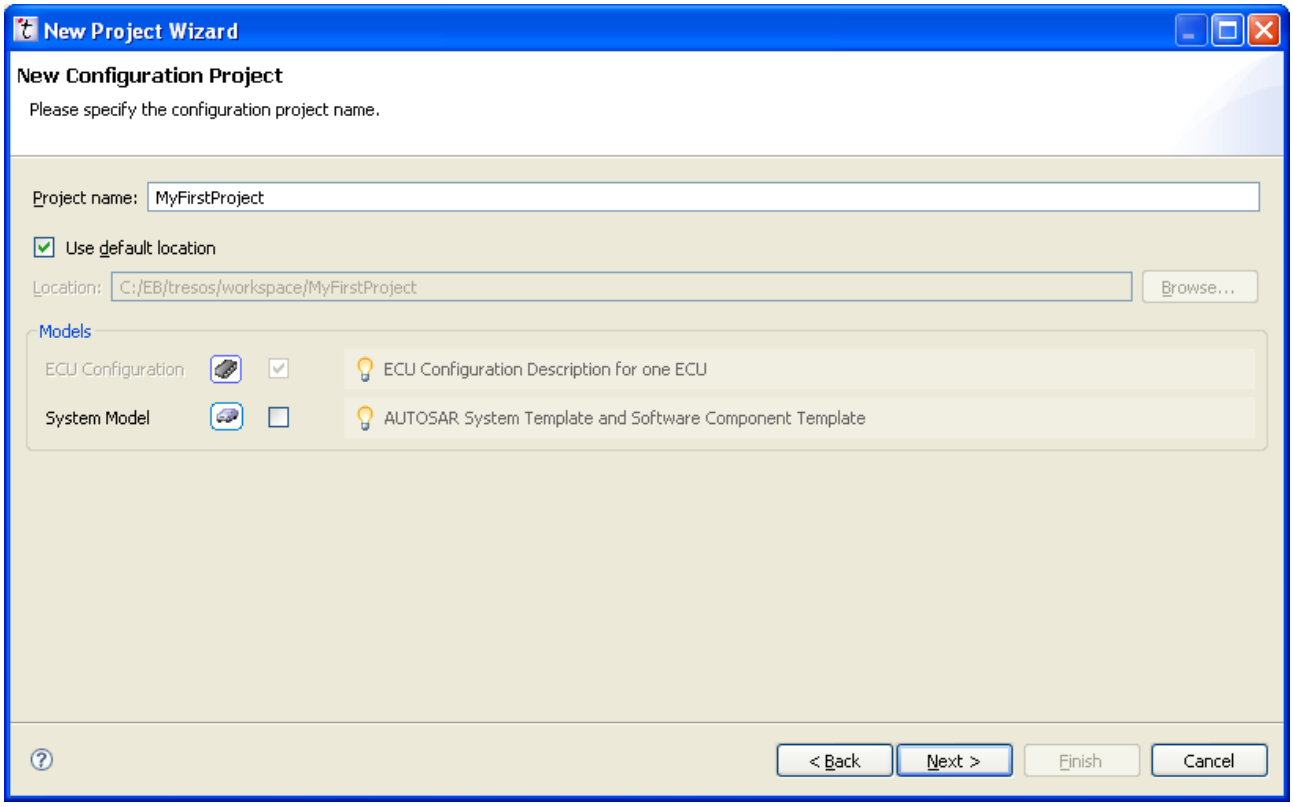
- Select *Project*.
- Select *Configuration Project* from the list.

> **Note**
>
> The *Configuration Project from existing project* option is used to upgrade an existing project to a newer Autosar version. Details about this can be found in Section 4.8.1: Converting a project to a new Autosar version.

- Select **Next**.

  The *New Configuration Project* window opens up.

4.6.1.2. Step 2: Creating a new project



- Enter a project name in the *Project name* text control.

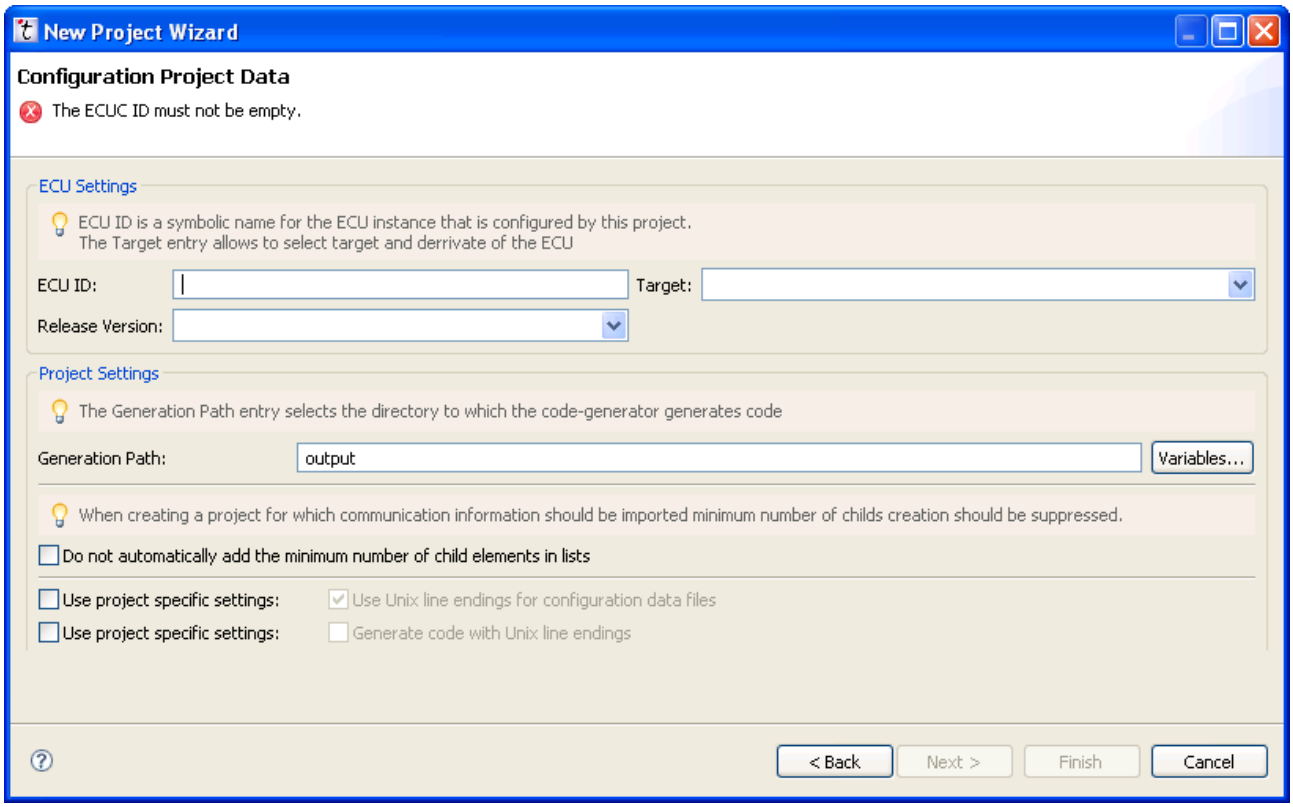
  Note

  The project name must be a valid Autosar shortname.

- To store the project in the workspace folder, check the box **Use default location**.
- To store the project in another location, uncheck the checkbox and select a path.
- Select **Next**.

  The *Configuration Project Data* window opens up.

4.6.1.3. Step 3: Configuring project properties



In the Configuration Project Data window:

- Fill in the *ECU ID*.

  Most likely this ID is provided by the OEM as part of a system description that describes a vehicle as a whole.

- Fill in the *Target*. Use the drop down listbox to select the target/derivate of your ECU.
- Fill in the *Release Version*. Use the drop down listbox to select the Autosar release version of your project.

  For example, if you select Autosar 2.1, you can add only modules with release version 2.1x to your project. If you select Mixed, you may add all available modules for the selected target/derivate and configure module configurations with different release versions.

> **Note**
>
> This choice affects the Exporter and restricts the possible target content types (see Section 4.11.2.2: Importing and exporting configuration data from and to the Autosar format).

- Select the **Project Settings** according to your preferences.

- To automatically add the minimum number of child elements in lists, keep the checkbox **Do not automatically...** empty.
- To use project-specific settings, check the checkbox **Use project-specific settings**.
- *Line endings of configuration data files:* By default, the type of line endings which is used for the files which store the configuration data is controlled by a global workspace setting (see Section 4.5.3.3: Setting configuration preferences). If you check the checkbox **Use project-specific settings**, the local setting takes effect for the project which you are just creating. You can then check or uncheck the checkbox **User Unix line endings ...**. The value you choose will only be active for this project.

  Note

  You can change this setting later after the project is created via the project preferences (see Section 4.6.7.2: Editing the Code Generator settings).

- *Line endings of generated code:* By default, the type of line endings of the files generated by the code generator (see Section 4.10: Generating code for projects) is controlled by a global workspace setting (see Section 4.5.3.3: Setting configuration preferences). If you check the checkbox **Use project-specific settings**, the local setting takes effect for the project which you are just creating. You can then check or uncheck the checkbox **Generate code with Unix line endings** and this setting will only be active for this project.

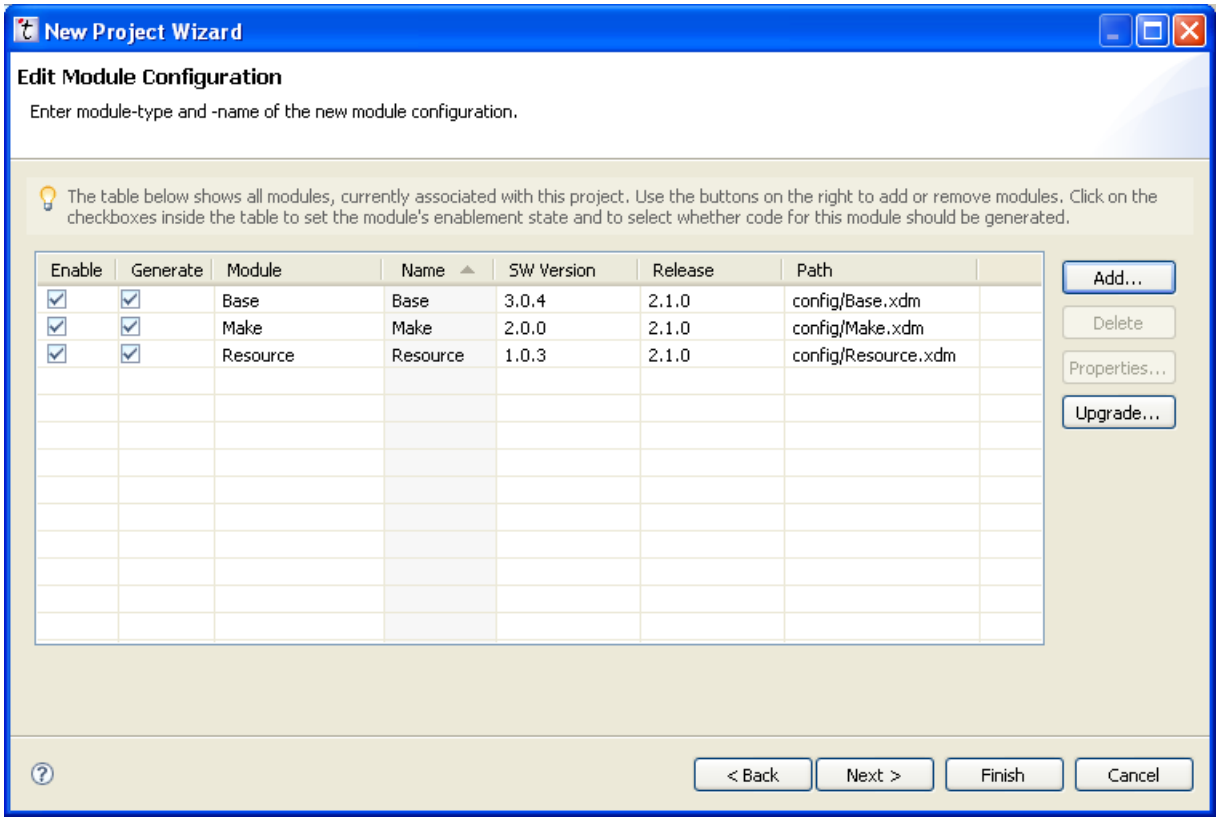
  Note

  You can change this setting later after the project is created via the project properties (see Section 4.6.7.2: Editing the Code Generator settings).

- Select **Next**.

  The *Edit Module Configuration* page opens up.

4.6.1.4. Step 4: Defining module configurations



The *Edit Module Configuration* page lets you add or remove the module configurations of the project you want to create. Some modules are mandatory for your architecture or Autosar release version. These modules are added automatically and are already present when you open the page. It is possible to remove them, but it is not recommended.

The features of this wizard page are explained in the following subsections. After you are done setting up your initial module configurations, press the **Next** button and continue reading at the next section.

Note

You can also create a project which initially contains no module configurations and add them later.

4.6.1.4.1. Adding the module configurations

To add new module configurations:

- Press the **Add** button.

  The *Add Module/Edit Module Configuration* window opens up. The options for adding a module are described in Section 4.7.3: Adding new module configurations.

- When the *Add Module/Edit Module Configuration* window is closed, you can see the *Edit Module Configuration* wizard page again. You can repeat this step as often as you need to add further module configurations.

4.6.1.4.2. Editing the module configuration

To edit the module configuration:

- Select the **Properties** button.

  This opens up the *Module Configuration* wizard for the selected configuration.

4.6.1.4.3. Deleting module configurations from the list

To delete any of the module configurations:

- Select **Delete**. The configurations selected are removed from the list.

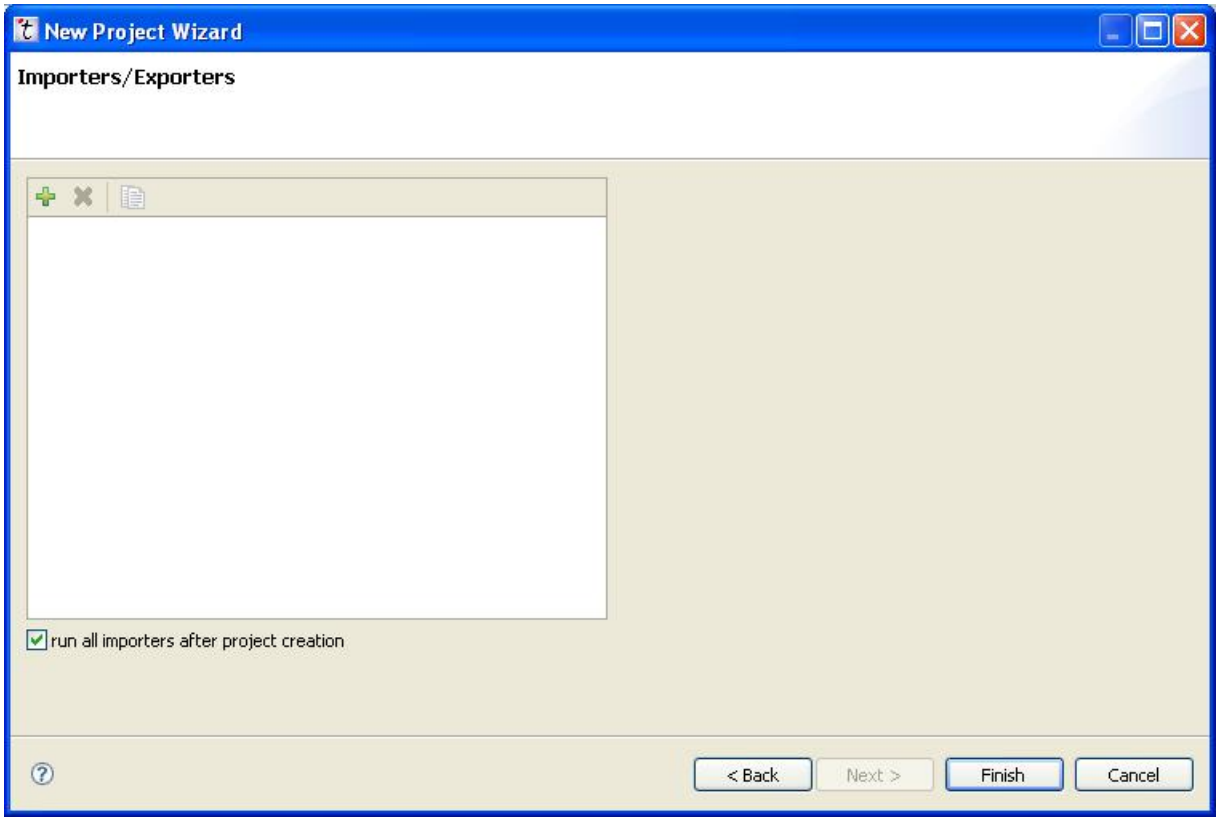
> **Note**
>
> The deletion does not affect any config files at this step because the project has not yet been created.

4.6.1.4.4. Upgrading module configurations

The **Upgrade** button lets you upgrade a module from an older software version of the module to a newer one. See Section 4.8.2: Upgrading module configurations to new module versions for details.

4.6.1.5. Step 6: Importing data into the project (optional)

It is possible to import configuration data into the project directly when it is created. This step is optional, you can also import the data later, or not at all. See Section 4.11.2: Importing and exporting configuration data for details on how to do this. If you want to skip this part, continue directly to the next section.



4.6.1.6. Step 7: Finishing set-up of the new project

To finish setting up the new project:

- Click the **Finish** button. The wizard closes an you see the main window again. If you are creating a project with a lot of modules or if you are importing big files the creation may take a while and a dialog with a progress bar may appear. If the work is completed, the progress bar dialog closes.

  The new project is now visible in the *Project Explorer*. It is automatically loaded and you can start editing the module configurations.

4.6.2. Saving a project

If you have changed a project, e.g. by editing one of the module configurations, you need to save the changes to disk. Each module configuration is saved to a separate file. The location of the file has been set when the module was added to the project (see Section 4.7.3: Adding new module configurations).

To save a project

- Select the project node in the Project Explorer.

- Right-click on the node.

  A context menu opens up.

- Select **Save**

4.6.3. Renaming a project

If you want to change the name of a project after it has been created, you can rename it.

To rename a project

- Select the project node in the *Project Explorer*.
- Right-click on the node.

  A context menu opens up.

- Select *Rename* and type in a new name.

  The project now appears in the project explorer under its new name.

4.6.4. Copying a project

To copy a project:

- Right-click on the project node you want to copy.

  A context menu opens up.

- Select the *Copy* entry

  The project is copied to the clipboard.

To paste the project under a new name:

- Right-click in the *Project Explorer*.

  A context menu opens up.

- Select the *Paste* entry.

  A dialog opens up and prompts you for a name of the new project. Since the project name must be unique you have to decide on a new name. After entering the name press the **OK** button.

  The dialog closes and the new project is visible in the *Project Explorer*.

Warning

If you manually changed any paths of the project to absolute paths, both projects now work on the same directories. This may be the case if you changed e.g. the output directory, or the location of the config files during project creation or in the project properties. If you did not change anything, only directories relative to the project directory are used and both project are completely separated from another.

4.6.5. Deleting a project

It is possible to delete a project. Deleted projects cannot be opened again.

To delete a project

- Select the project node in the *Project Explorer*.
- Right-click on the node.

  A context menu opens up.
- Select *Delete*. A dialog opens up asking you if you really want to delete the project. Additionally, there are two radio buttons in the dialog which let you make a choice:
  - if you decide to *also delete contents ...*, the project will completely be removed from disk. The disk space will be freed, but you will have no way of restoring the project.
  - if you decide to *do not delete contents*, all associations to the project inside EB tresos Studio are removed, but the project remains on disk (inside the `workspace` directory). If you want to restore the project you can do so by importing it again into EB tresos Studio (see Section 4.11.1.1: Importing a project).

4.6.6. Closing and opening projects

By default the projects are always open in EB tresos Studio. If you are working with serveral very big projects and if you want to reduce the memory requirements or the speed of EB tresos Studio you can close the projects that you currently do not need.

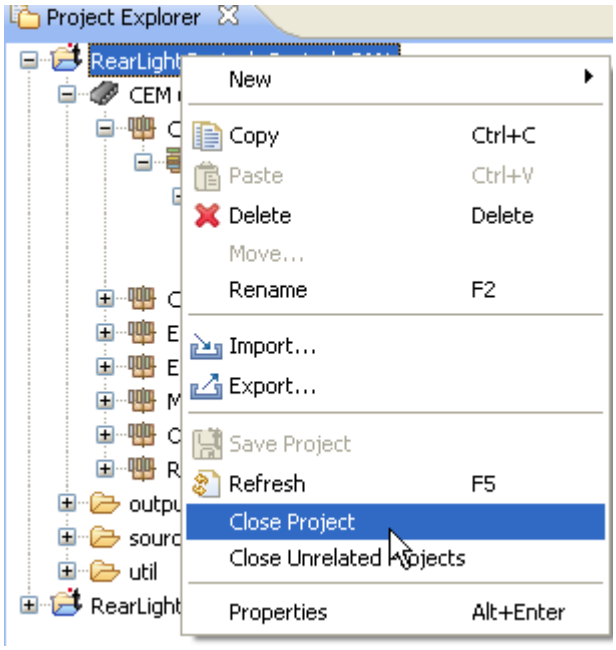
Closed projects only use very few resources, its content is not loaded and you cannot edit their configurations or generate code for them. A closed project needs to be opened and then loaded to be fully functional. More information about the state of a project can be found in Section 4.4.3.1: Project and ECU configuration states.

4.6.6.1. Closing projects

To close a project:

- Select the project node in the *Project Explorer* .
- Right-click on the project node.

  A context menu opens up.



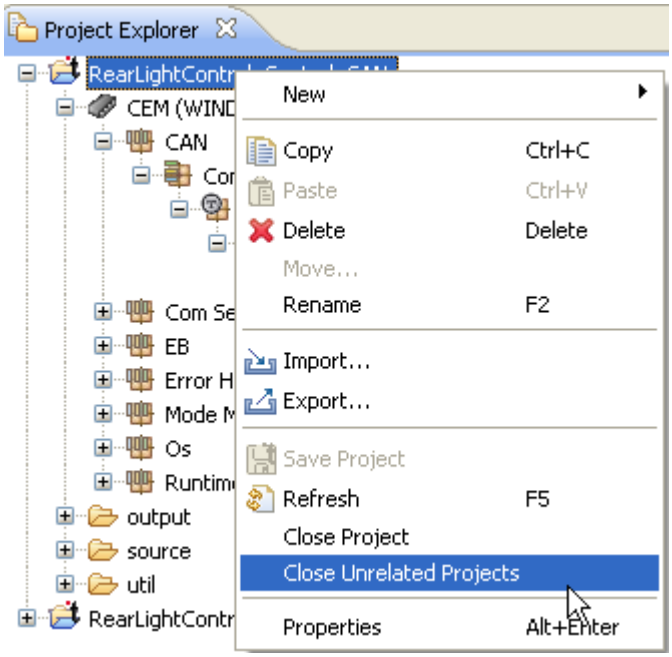
- Select *Close Project*

  The project selected is being closed.

To close all projects that are not referenced by your current project:

- Select the project node you want to keep open in the Project Explorer.
- Right-click on the node.

  A context menu opens up.



- Select *Close Unrelated Projects*.

  All projects that are not referenced by your current project are closed.

4.6.6.2. Opening a project

To open a closed project:

- Right-click on the closed project to open the *Project Explorer* 's context menu.
- Select **Open Project**.

  The project is now open but yet unloaded.

4.6.6.3. Loading a project

To load an open project take one of the following two steps:

- Alternative 1: Double-click on the ECU configuration node

  The project starts loading its configuration.

- Alternative 2: Right-click on the ECU configuration node to open the Project Explorer's context menu, then select *Load Configuration*

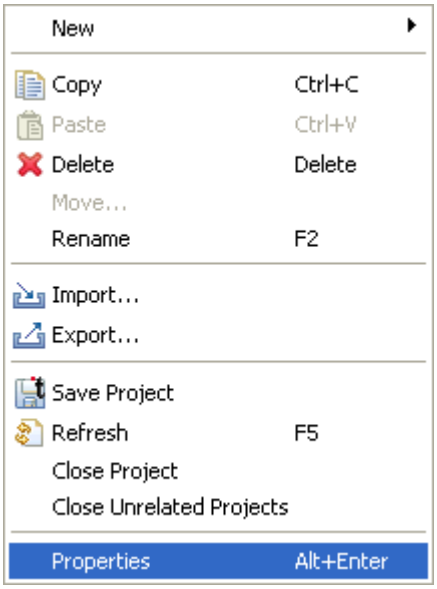
  The project starts loading its configuration.

4.6.7. Viewing and changing project properties

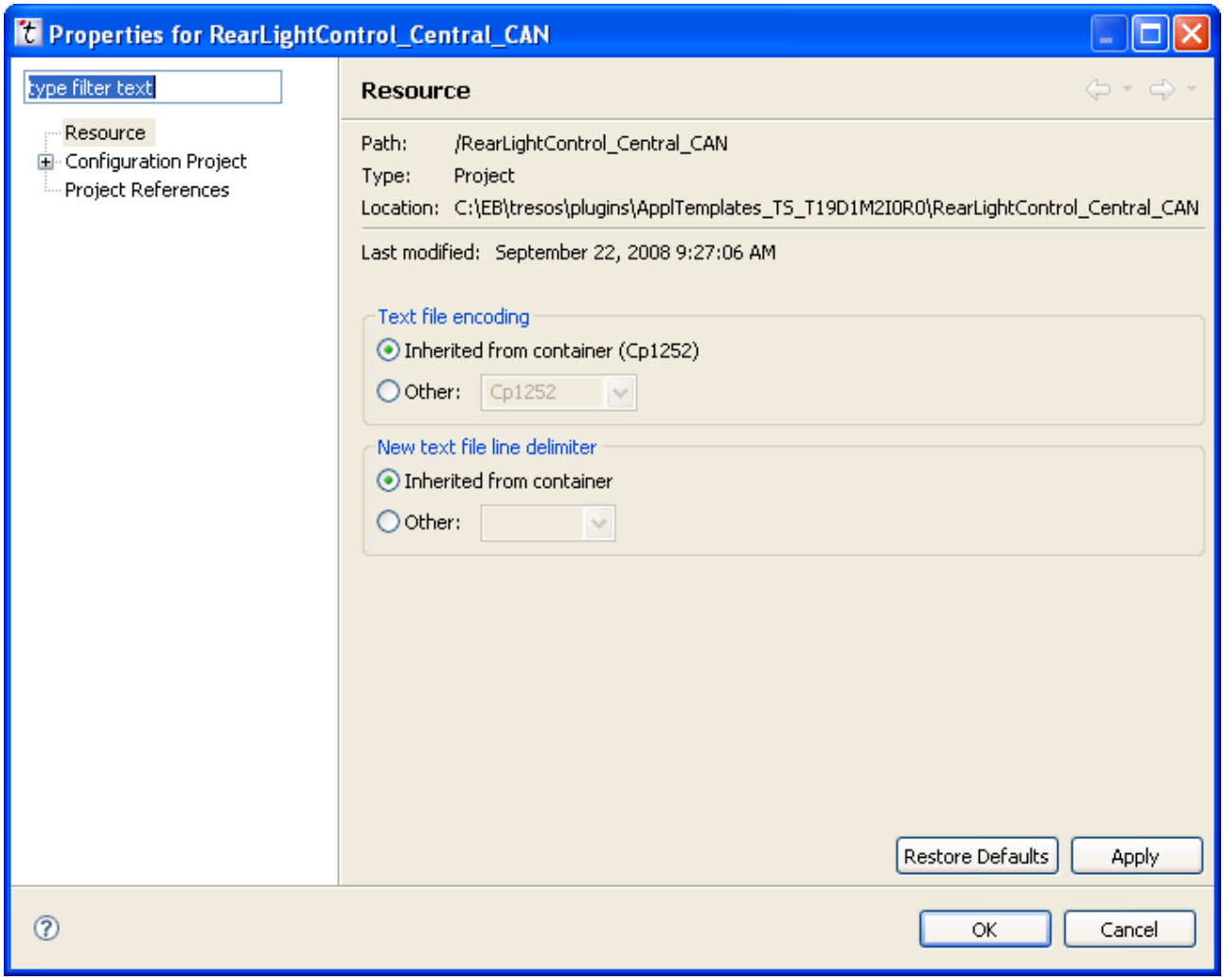
The data you have entered in the *New Project* wizard can be viewed and changed. This is done in the *Properties for project name* dialog. To open this dialog:

- Select the project in the Project Explorer.
- Right-click on the project.

  A context menu opens up.
- Select *Properties* from the menu.



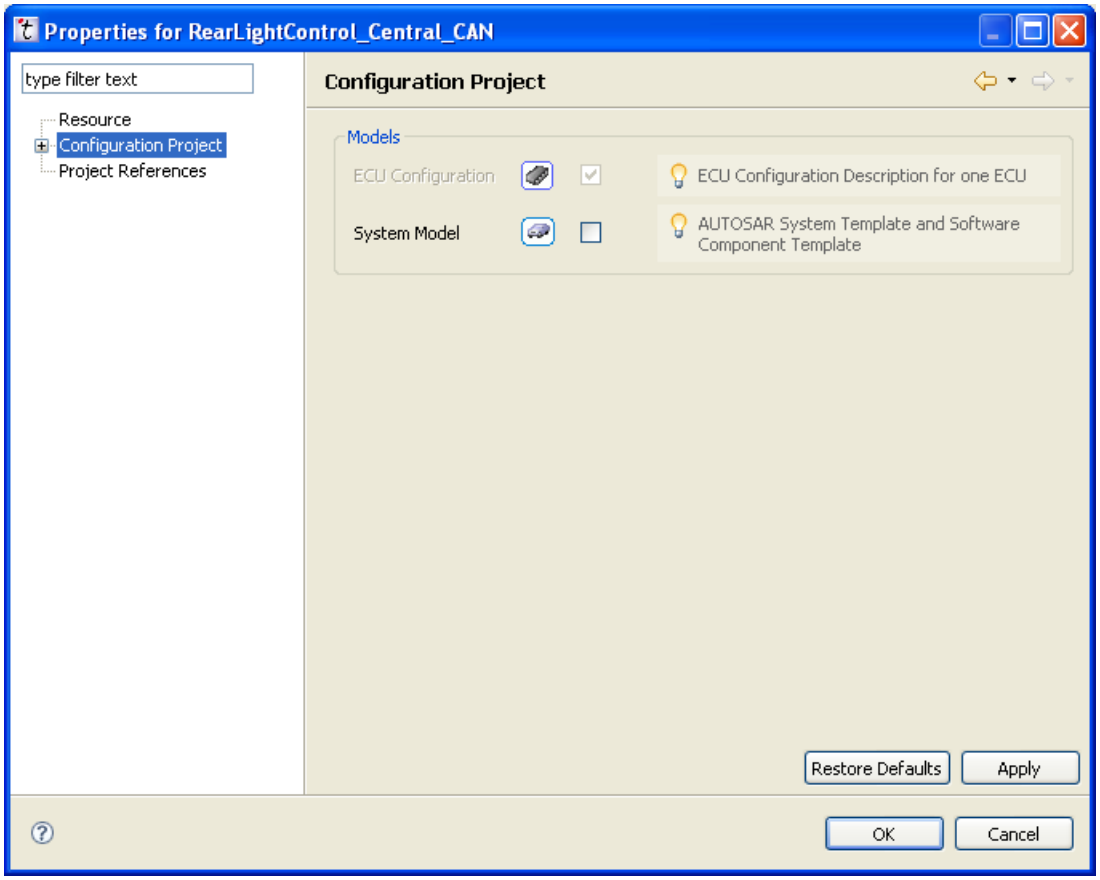
The *Properties for project name* dialog opens up.



The *Properties for project name* dialog consists of a tree on the left and a main area. The available properties are divided into different categories which make up the items of the tree on the left. Some categories contain subcategories. If you choose a category by selecting an item of the tree, the properties of that category are displayed in the main area of the dialog. The categories which are relevant for the work with EB tresos Studio are grouped below *Configuration Project*. You can expand this tree item to see more categories.

4.6.7.1. Listing the supported models of the project

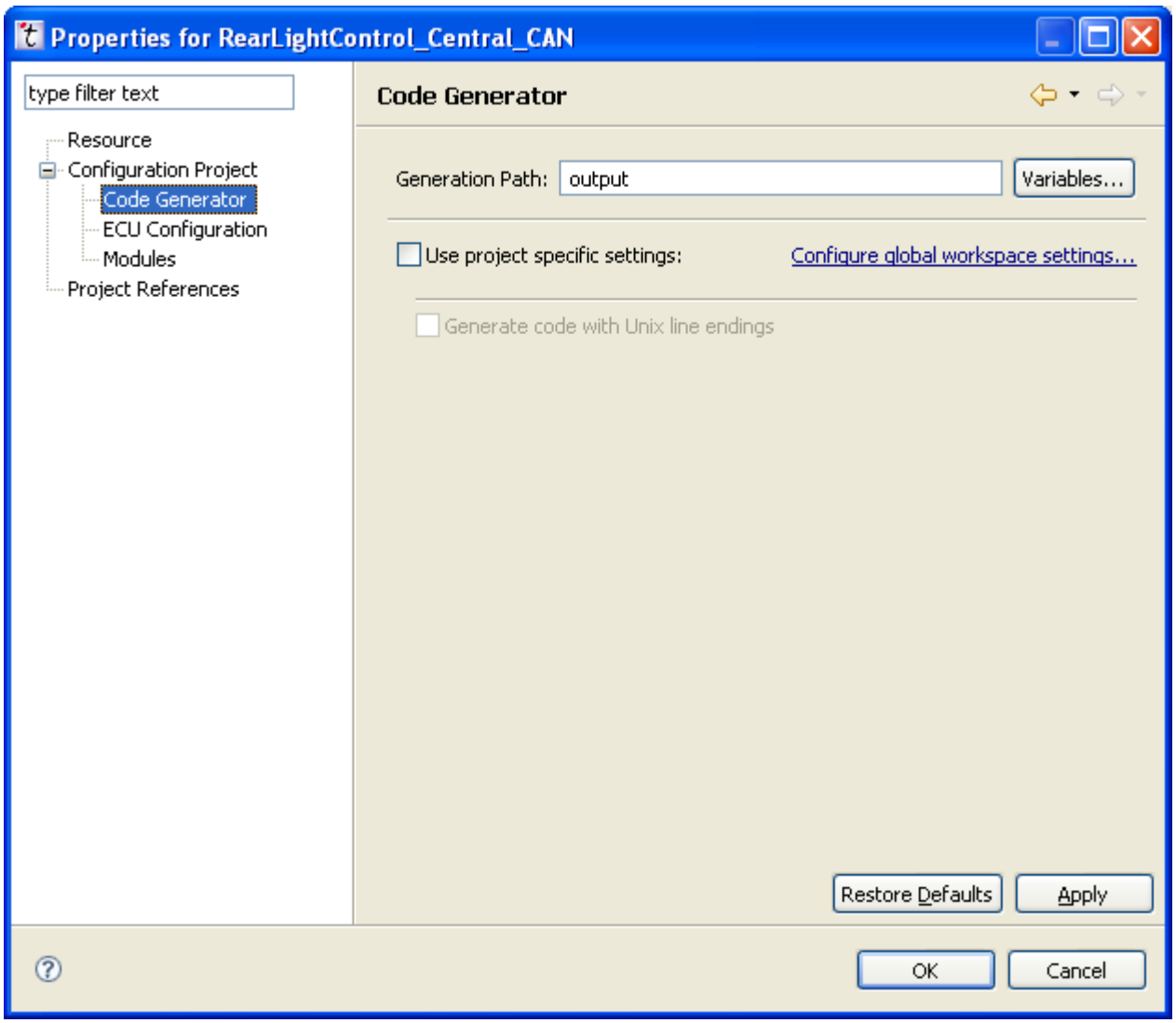
On the *Configuration Project* page of the project properties you can see which models are enabled in this project. The *ECU Configuration* model is always enabled. The *System Model* is automatically enabled when an RTE module is present in your project. You can also enable the *System Model* manually without an RTE module present, but currently this does not have any effect.



4.6.7.2. Editing the Code Generator settings

The properties on the *Code Generator* page affect the behaviour of code generators. To open this page:

- Select *Code Generator* from the tree view on the left.



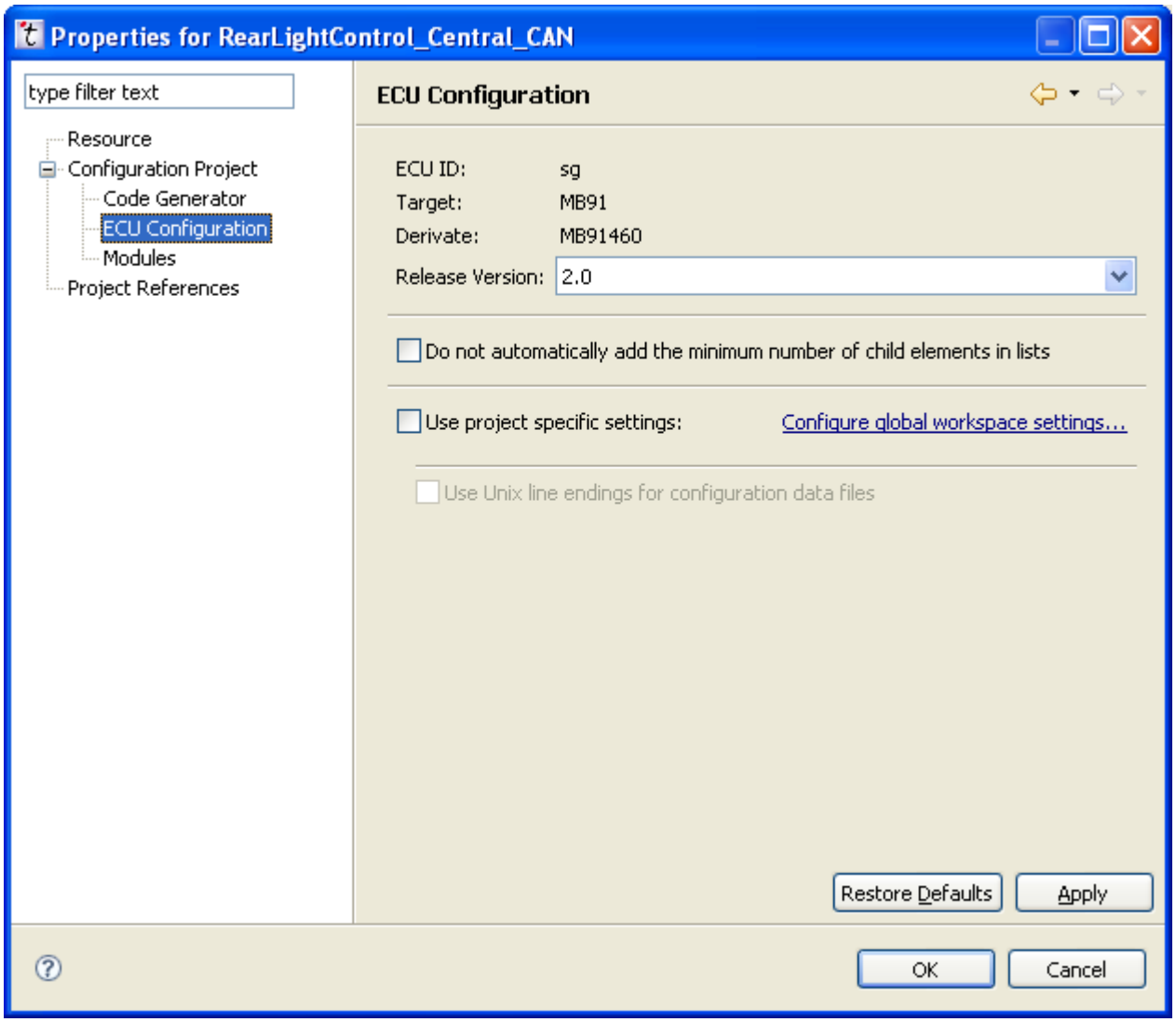
- The **Generation Path** textbox controls into which directory the generated code files are written to. If the value is a relative path, the path is considered to be relative to the project's directory. If you did not change the generation path during the creation of the project, the value is `output`. If you want to change the value, you may enter an absolute or a relative path. You can also use the predefined variables that are available by clicking the **Variables** button. There are variables defining the workspace directory, the tresos installation directory and so on.
- If the **Use project specific settings** checkbox is not set, the global workspace-wide value for this property is used. You can view and change this setting by clicking on the link *Configure global workspace settings* (see Section 4.5.3.3: Setting configuration preferences).
  - If you want to change the unix line endings for generated code just for this project:
    - Select *Code Generator* from the tree view.
    - Check the **Use project specific settings** checkbox.

- Check the **Generate code with Unix line endings** checkbox if you want to have Unix line endings in your generated files. Uncheck the checkbox if you want to have Windows line endings in your files.
- Select **Ok** to apply this for the current project.

4.6.7.3. Editing the ECU Configuration settings

The *ECU Configuration* page allows you to change settings that influence the way how ECU configurations are handeled by the tool. To open this page:

- Select *ECU Configuration* from the tree view on the left.



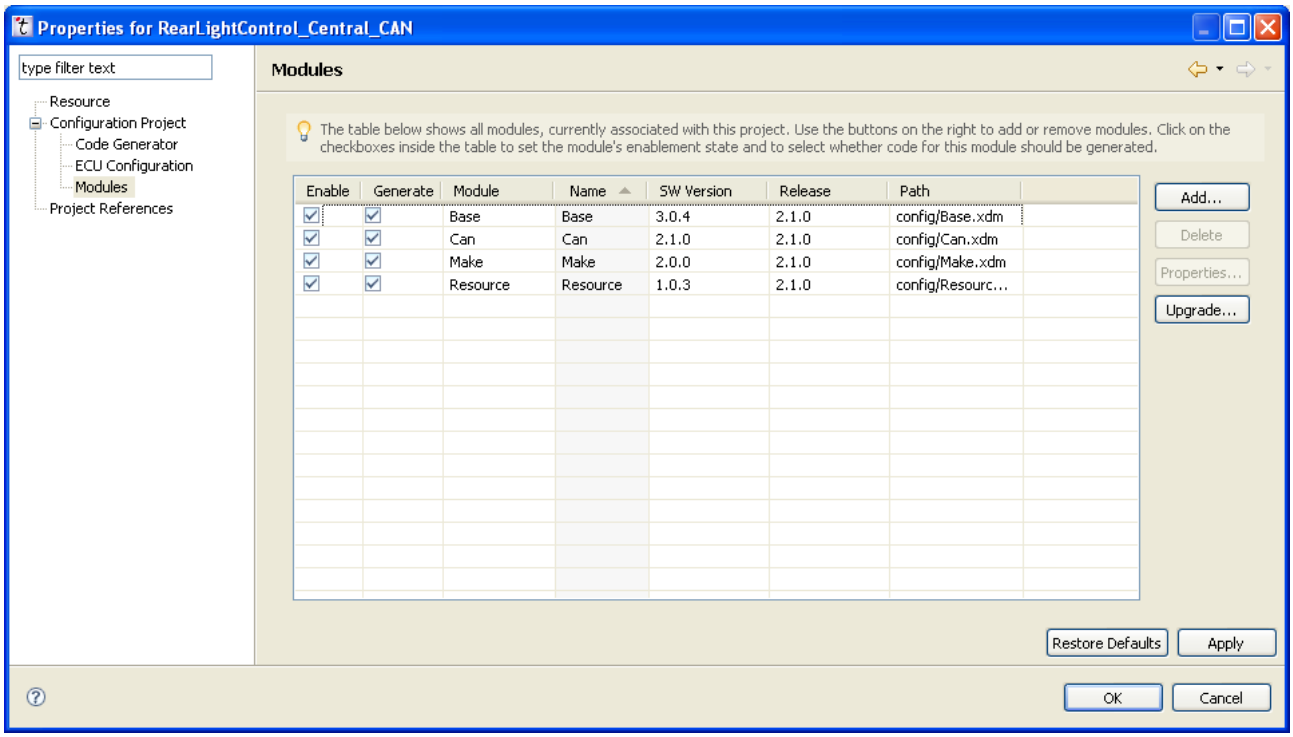
The main part of the dialog provides you with several options:

- You can view the *ECU ID*, *Target* and *Derivate* settings. These values have been set when the project was generated (see Section 4.6.1.3: Step 3: Configuring project properties and cannot be changed.
- The **Release Version** drop down listbox lets you view the Autosar release version of your project. If you are not working with *mixed* release versions in your project, you cannot change this value.
- If you do not want lists to be filled with default entries, check the **Do not automatically add…** checkbox. If the checkbox is unchecked, the list elements will be added automatically.
- You can change which line endings the configuration data files use. If the **Use project specific settings** checkbox is not set, the global workspace-wide value for this property is used. You can view and change this setting by clicking on the link *Configure global workspace settings* (see Section 4.5.3.3: Setting configuration preferences).

  To change the setting for Unix line endings for configuration data files in this project independently of the global settings in EB tresos Studio, check **Use project specific settings**. Then change *Use Unix line endings for configuration data files* to the value you would like to have in this project.

4.6.7.4. Viewing information about the project's modules

In the *Modules* page of the project properties, you may configure or edit a module configuration. See details in chapter Section 4.7.3: Adding new module configurations.



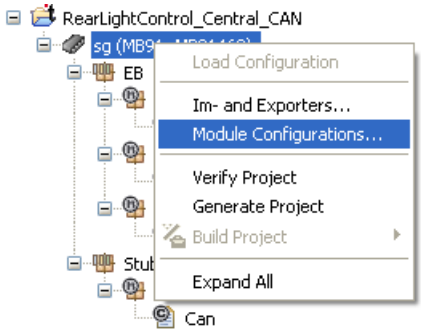
4.7. Working with ECU configurations

This section deals with the management of ECU module configurations. It explains how to add or remove configs, how to find detailed informations about the module configurations of a project, and how to enable of disable them.

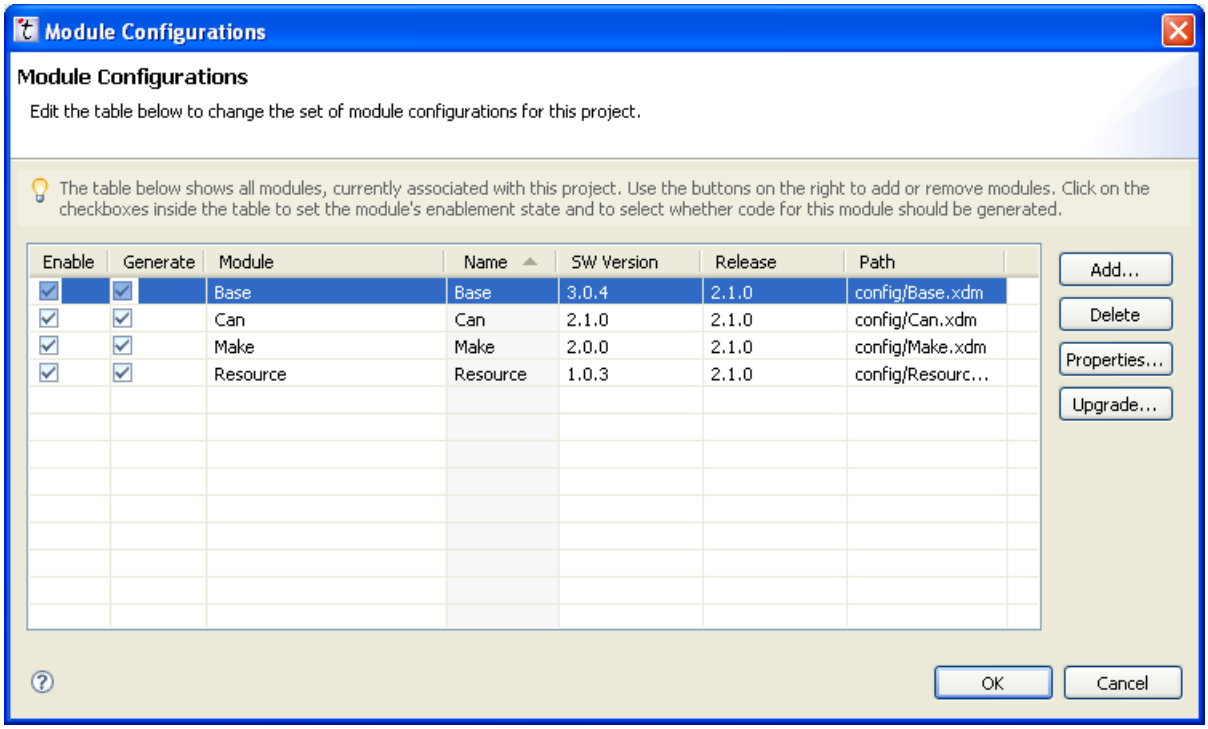
4.7.1. Viewing the list of module configurations for a project

To change the set of module configurations for a particular project:

- Select the ECU configuration node within the project.
- Right-click on the ECU configuration node.
- Select **Module Configurations...**.



The window *Module Configurations* opens up.



4.7.2. Viewing detailed information of a module configuration

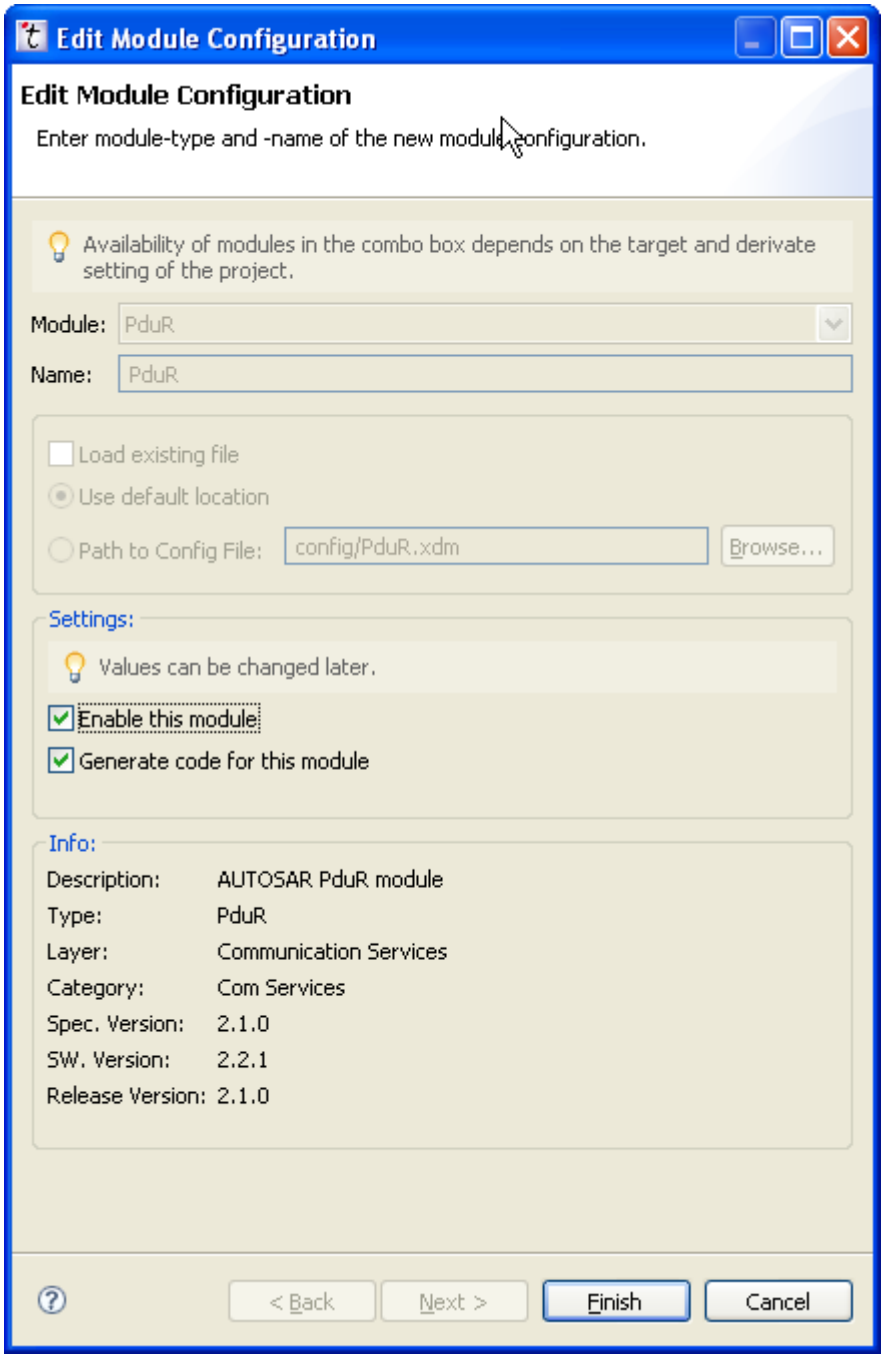
To view more information about or to edit an existing module configuration in the *Module Configurations* dialog:

- Alternative 1: Double-click on the module config inside the list

  The *Edit Module Configuration* opens up for the selected configuration.

- Alternative 2: Select the module config you want to edit, then click the **Properties** button.

  The *Edit Module Configuration* opens up for the selected configuration.



This dialog is similar to the *Add Module* (see Section 4.7.3: Adding new module configurations). the difference is that the GUI elements which cannot be changed for already existing module configurations are disabled.

- To enable a module configuration file, check the **Enable this module** checkbox.

See details about enabling and disabling a module configuration in chapter Section 4.7.6: Disabling and enabling module configurations

- To enable the code generation for the module, check the **Generate code for this module** checkbox.

See details about enabling code generation in chapter Section 4.7.5: Disabling code generation of a module configuration.

- Select **Finish**.

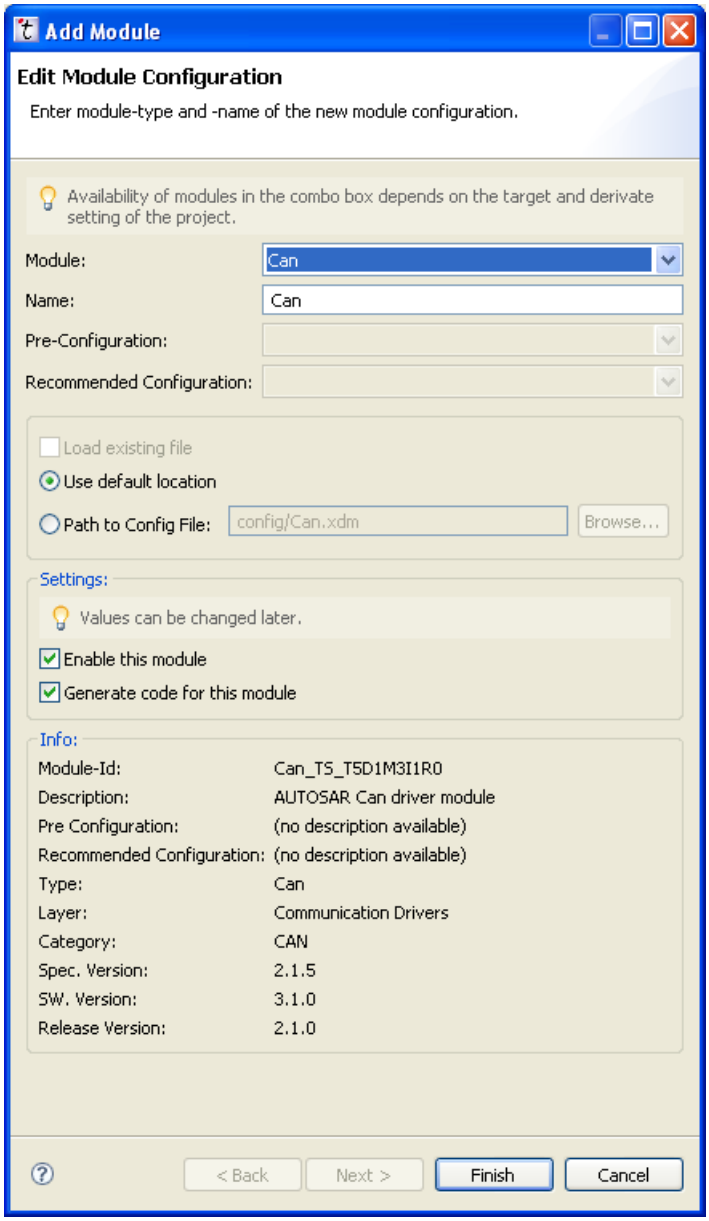
  The *Edit Module Configuration* window closes.

4.7.3. Adding new module configurations

To add new module configurations in the Module Configurations window:

- Click the **Add** button.

  The *Add Module/Edit Module Configuration* dialog opens up. This dialog may consist of multiple pages.



- Select a module from the *Module* drop down listbox.

  The modules available depend on the configured target/derivate and the EB tresos Studio release version.
- Enter a name for the module configuration in the *Name* textbox
- If the module supports pre-configurations, you must choose a pre-configuration in the *Pre-configuration* dropdown listbox. The pre-configuration contains values for configuration parameters. It is loaded directly after the module is added to the project and the parameters contained it are not editable any more.
- If supported by the module, you can choose a recommended configuration in the *Recommended Configuration* dropdown listbox. A recommended configuration contains values for configuration parameters. The values can be considered as a guideline; they may be changed later by the user.

- To create a new module based upon the configuration input of an existing file, check the *Load existing file* checkbox.

  The checkbox is grayed out if the entered filename (or default location) does not exist.

  Warning

  Any existing file with that name will be overwritten when you create the module, unless you check the box.

- To use the default location for the configuration file that stores the module configuration, select *Use default location*. By default, the module configurations are stored in a folder called `config` inside the project directory. The name of the file corresponds the name of the module configuration.
- To enter a different location for the configuration file, select *Path to Config File*. The text control becomes enabled and you can type in a different name and path. You can open a file browser by clicking the **Browse** button.
- To enable the module configuration, check the *Enable this module* checkbox. If the module configuration is disabled, you cannot edit it and no entries in the problems view are shown for it.
- To enable code generation for this module, check the *Generate code for this module* checkbox.

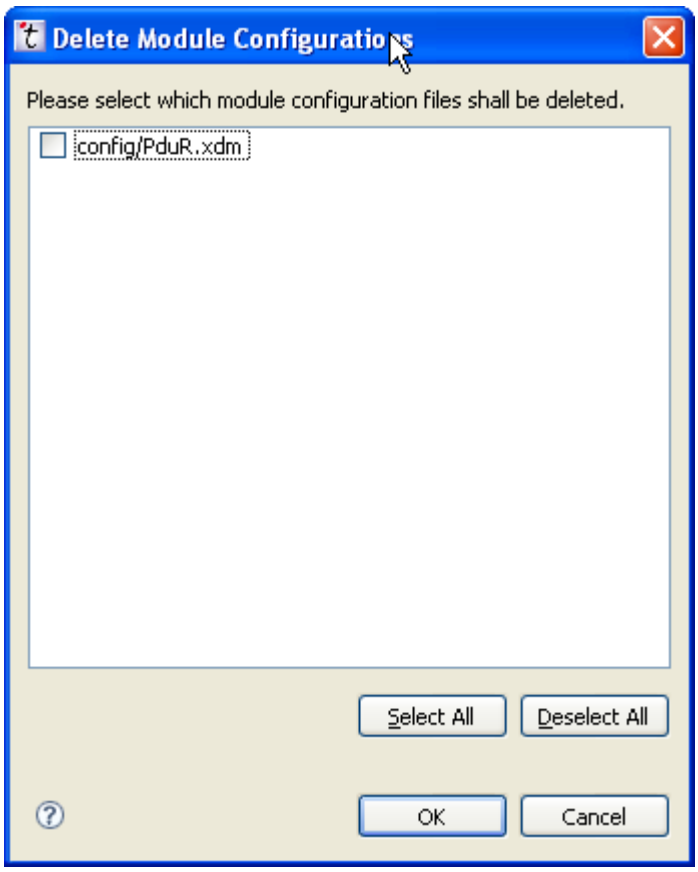
  Alternatively, you may enable code generation in the *Module Configurations* window (see Section 4.7.5: Disabling code generation of a module configuration).
- Select **Finish** and repeat this step as often as you need to add further module configurations.

4.7.4. Deleting module configurations

To delete any of the module configurations in the *Module Configurations* dialog:

- Click the **Delete** button.

  You can choose if you just want to remove the module configuration from the project, or if you also want to remove the configuration data files from disk. When you click the **Delete** button, the *Delete Module Configurations* dialog opens up.



This dialog lets you choose which configuration data files you want to remove from disk and which ones you want to keep. The module config is removed from the project in any case.

To schedule a file for deletion:

- Check the checkbox in front of the configuration file you wish to delete.
- Select **Ok**.

> **Warning**
>
> This action deletes the existing configuration file, it cannot be restored!

The configurations selected are removed from the list.

To finish adding or deleting module configurations in the *Module Configurations* window:

- After you are done changing the modules, press the **Finish** button and all changes are executed.

4.7.5. Disabling code generation of a module configuration

To enable code generation of a module configuration:

Enable	Generate	Module	Name	Path	Release	Description
☑	☑	Base	Base	config/Base.xdm	2.1.0	Base Bundle of the EB tresos Au
☐	☑	CanIf	CanIf	config/CanIf.xdm	2.1.0	AUTOSAR CanIf module
☑	☑	Com	Com	config/Com.xdm	2.1.0	AUTOSAR Com Module
☑	☑	Det	Det	config/Det.xdm	2.1.0	AUTOSAR Development Error Tr
☑	☑	EcuC	EcuC	config/EcuC.xdm	2.1.0	AUTOSAR EcuC module
☑	☑	EcuM	EcuM	config/EcuM.xdm	2.1.0	AUTOSAR EcuM State Manager
☑	☑	Make	Make	config/Make.xdm	2.1.0	Application build environment
☑	☑	OS	Os	config/Os.xdm	2.1.0	OS (WINDOWS/WIN32X86)
☑	☑	PduR	PduR	config/PduR.xdm	2.1.0	AUTOSAR PduR module
☑	☑	Rte	Rte	config/Rte.xdm	2.1.0	AUTOSAR Runtime Environment
☑	☑	SchM	SchM	config/SchM.xdm	2.1.0	AUTOSAR Schedule Manager

- Select the module configuration you wish to enable or disable code generation for.
- Check the **Generate** checkbox, if you wish to enable code generation for this module configuration.
- Uncheck the **Generate** checkbox , if you wish to disable code generation for this module configuration.

4.7.6. Disabling and enabling module configurations

EB tresos Studio allows you to temporarily disable configurations. Disabled configurations are treated as if they do not exist, which means you cannot edit or verify them, and code generators cannot access them.

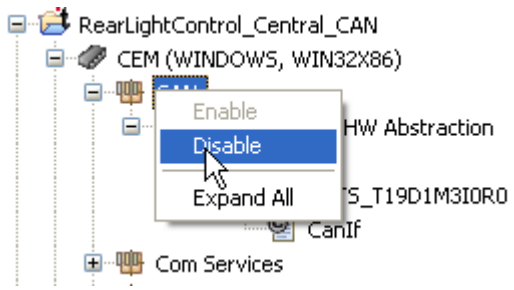
4.7.6.1. Disabling module configurations

There are four different alternatives to disable module configurations.

Alternative 1: To disable one or more module configurations beneath a category node in the *Project Explorer*:

- Select the category node in the *Project Explorer*
- Right-click on the category node

  A context menu opens up.



- Select *Disable (All)* from the context menu

The module configurations beneath the category node now are all grayed out and disabled.

The *Disable* menu item is only available if at least one enabled module configuration is available beneath the category node. If just one is available, the menu option reads *Disable*; if more than one is available, the submenu reads *Disable All*.

Alternative 2: You may disable individual module configurations from the module or the module configuration itself:

- Select the module or the module configuration you want to disable.
- Right-click on the module or module configuration.

  A context menu opens up.

- Select *Disable* from the context menu.

  The module configuration of that module is now disabled and grayed out.

Alternative 3: you can edit the module configuration list in the the *Project Properties* window. Please refer to the chapter Section 4.6.7.4: Viewing information about the project's modules.

Alternative 4: you can use the module configuration list which is avialable through the *Module Configurations...* menu item in the project explorer (see Section 4.7.1: Viewing the list of module configurations for a project).

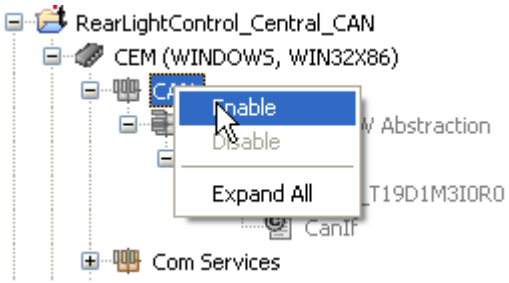
4.7.6.2. Enabling module configurations

There are four alternative ways to enable module configurations.

Alternative 1: In the Project Explorer:

- Select the category node
- Right-click on the category node

  A context menu opens up.



- Select *Enable (All)* from the context menu.

  The module configurations beneath the category node now are all displayed in color and are enabled again. The *Enable* item is only available if at least one enabled module configuration is available beneath the

category node. If just one is available, the option reads *Enable*; if more than one is available, the option reads *Enable All*.

Alternative 2: You may enable individual module configurations from the module or the module configuration itself:

- Select the module or the module configuration you want to enable.
- Right-click on the module or module configuration.

  A context menu opens up.
- Select *Enable* from the context menu.

  The module configuration of that module is now enabled and displayed in color.

Alternative 3: you can edit the module configuration list in the the *Project Properties* window. Please refer to the chapter Section 4.6.7.4: Viewing information about the project's modules.

Alternative 4: you can use the module configuration list which is avialable through the *Module Configurations...* menu item in the project explorer (see Section 4.7.1: Viewing the list of module configurations for a project).

4.8. Upgrading projects and configurations

This chapter covers several aspects of working with projects and modules of different versions.

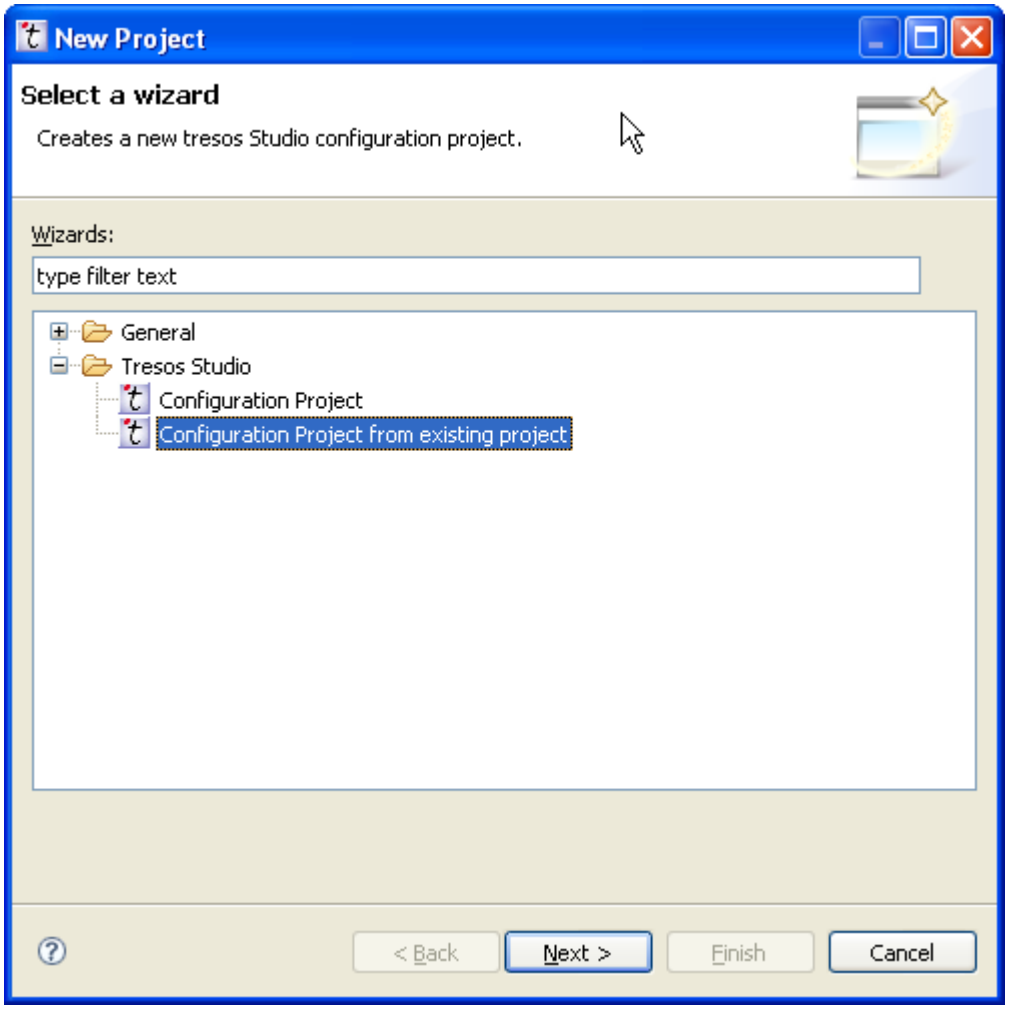
When a project is created, you have to decide on an Autosar release version for the project. It may happen at a later point in your development cycle that you want to migrate your project to a newer Autosar version. Section 4.8.1: Converting a project to a new Autosar version link describes how EB tresos Studio can help you with this.

Regardless of the Autosar release version number, the modules themselves contain a version number, called the *software version*. It is possible that you have installed modules of the same Autosar version, but with different software versions. If you have started a project with modules of an older software version, you may want to upgrade these modules to newer versions. For this case the *Module Configuration* dialog provides an upgrade function that is accessible via a button. The usage of this feature is explained in Section 4.8.2: Upgrading module configurations to new module versions.

4.8.1. Converting a project to a new Autosar version

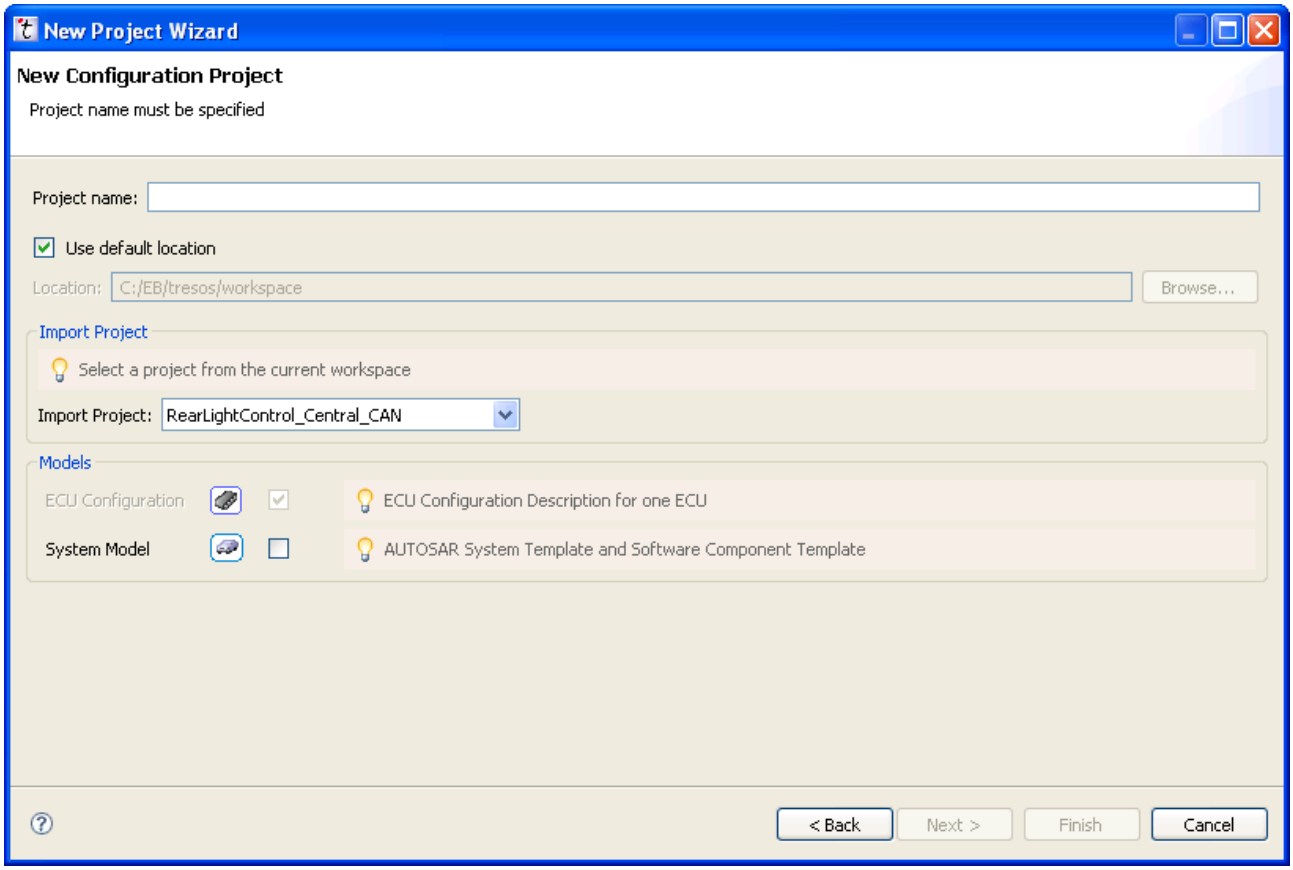
To convert a project (e.g. Autosar 2.0 configuration) to a newer version (e.g. Autosar 2.1 configuration):

- Follow the step 1 of Section 4.6.1.1: Step 1: Starting the Project Wizard.
- In step 2, select *Configuration Project from existing Project*.



- Select *Next*.

  The *New Configuration Project* window opens up.

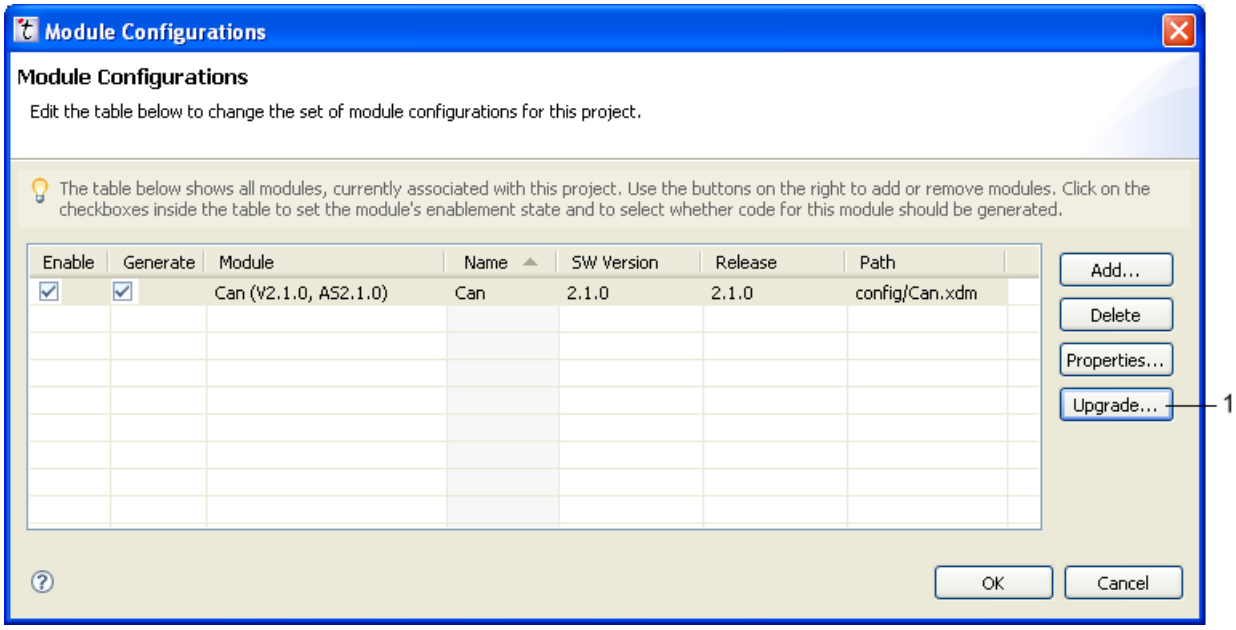


- Type in a name for the project in the *Project name* textbox.
- Provide the source project to copy from in the dropdown listbox *Import Project*.
- Proceed analog to (Section 4.6.1.2: Step 2: Creating a new project.

The dropdown listbox *Import Project* allows you to select just projects that are already present in the current workspace. If the project you would like to convert is not already in the workspace, please refer to chapter Section 4.11.1.1: Importing a project to import the desired project into the workspace. After you have imported the project into the current workspace, the *Import Project* dropdown listbox will automatically list the imported project as available.

4.8.2. Upgrading module configurations to new module versions

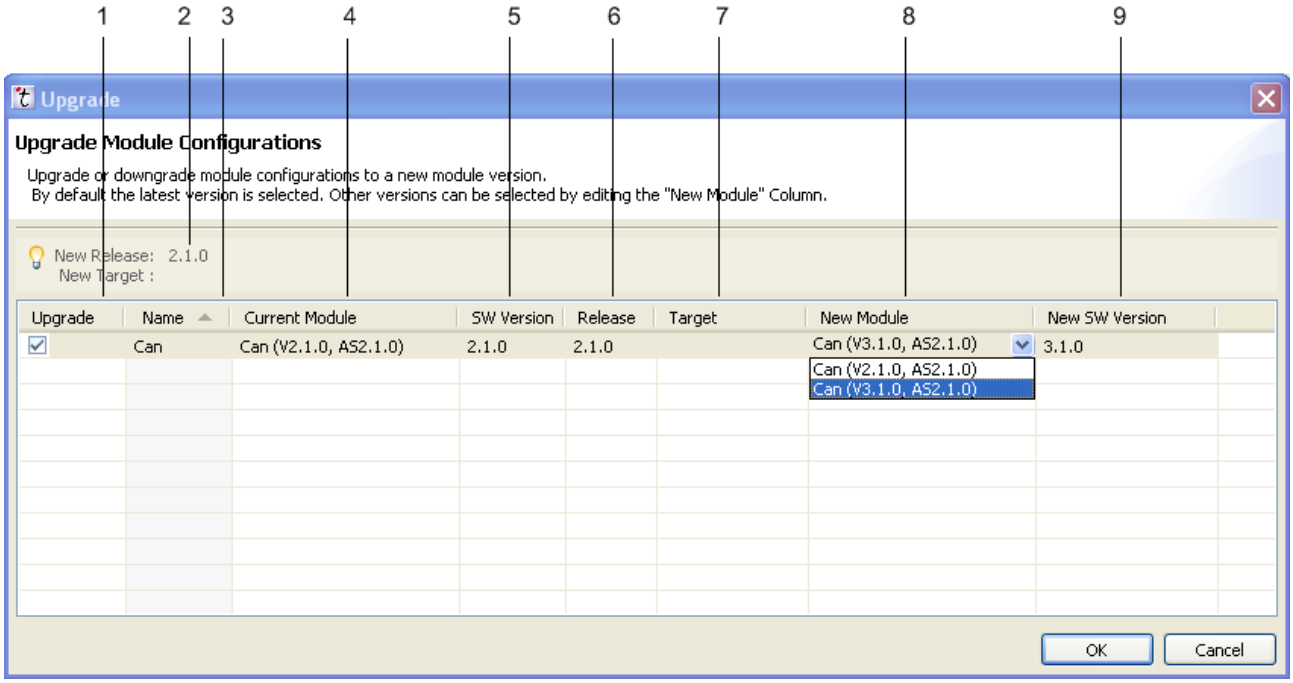
When new versions of modules are delivered and the configuration is changed, the new version is installed to EB tresos Studio as a new module with a new software version number. To switch a configuration between different versions of a module the Module Configuration dialog provides an upgrade function that is accessible via the button **Upgrade...**



The upgrade function is available from:

- The *New Project* wizard for new configuration projects (see: Section 4.6.1: Creating a new project).
- The *New Configuration Project from existing project* wizard for derived projects configuration projects (see: Section 4.8.1: Converting a project to a new Autosar version).
- The *Module Configuration* dialog accessible via the context menu of the *Project Explorer* (see: Section 4.7.3: Adding new module configurations).

If you click the **Upgrade** button, the *Upgrade* dialog is shown. When module configurations are selected in the *Module Configuration* dialog, only these are selected for upgrade, if no configurations are selected all module configurations are presented for upgrade.



Number	Description
1	Column *Upgrade*: The dialog allows to select the module configurations that are actually upgraded via the checkboxes in the column labeled *Upgrade*.
2	The info box shows release version and the target of the project which matches all presented new modules.
3	Column *Name*: This column shows the name of the module configuration.
4	Column *Current Module*: Shows the name of the module that is currently used for the configuration.
5	Column *SW Version*: Shows the software version of the module that is currently used for the configuration.
6	Column *Release*: Shows the release version of the module that is currently used for the configuration.
7	Column *Target*: Shows the target and derivate of the module that is currently used for the configuration.
8	Column *New Module*: Allows to select the new module to use for the configuration. By default the latest available is selected. Clicking into a call of this column presents a drop down box that allows to switch the module.
9	Column *New SW Version*: Shows the software version of the selected module.

Pressing the **Ok** button closes the dialog and updates the Module Configuration dialog. Closing the Module Configuration dialog actually switches the module configuration of the project to their new module definitions adapting configuration parameters if necessary.

4.9. Working with the ECU configuration editor

Each module that requires changes to its configuration provides one or more configuration editors. The configuration editor helps you editing the parameters of the module configuration by organizing them in an easily-understandable way by displaying additional information and by validating the entered values.

4.9.1. Editing a module configuration

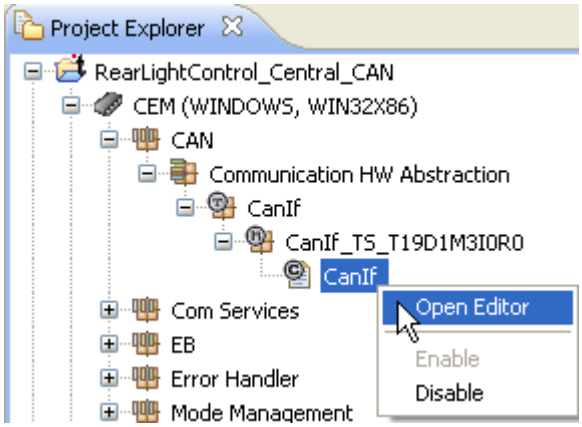
To configure or to edit a module:

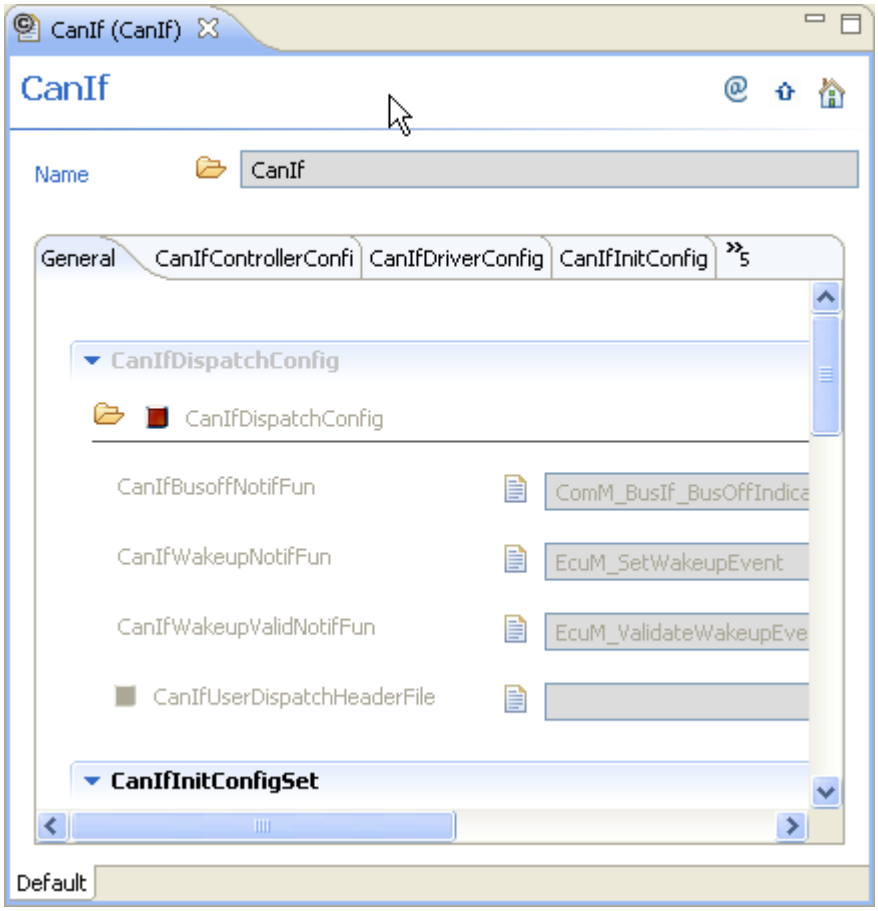
- Expand the tree underneath the project in the Project Explorer.

  The Project Explorer tree opens up to display several layers (depending on the Project Explorer display setting you use).

- Select the module configuration you want to edit. Either
  - double-click on the module configuration
  - or right-click on the module configuration and select *Open Editor*.

    This opens up the corresponding Editor view on the right hand side. This step will either open the Editor or bring the editor view to the front, if it has been opened before.



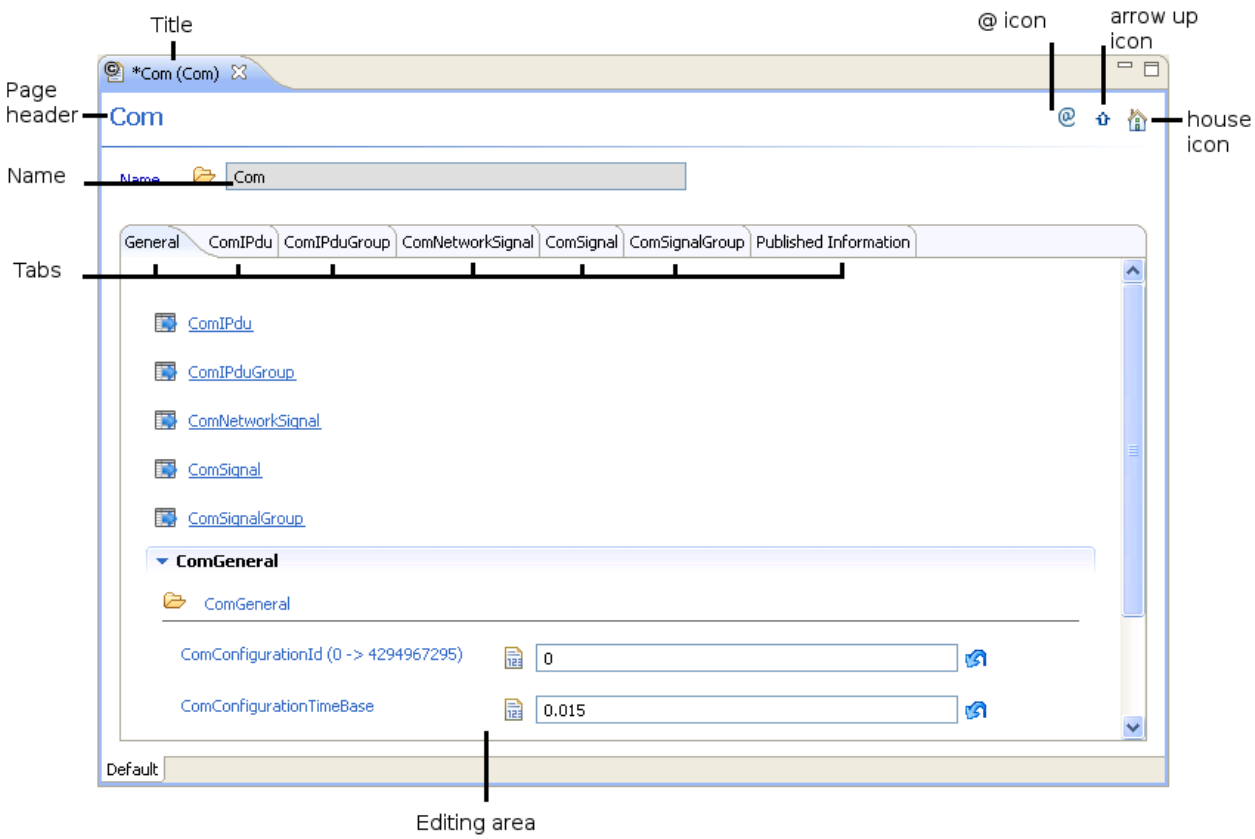


An Editor view of any module configuration may consist of several pages, which are displayed in tabs in the Editor view. Please refer to the chapter Editor view Section 4.9.2: Using the Editor view to get to know the set up of this view and its basic functionalities.

4.9.2. Using the Editor view

The *Editor* view is the central tool for editing the ECU configurations. The content of the view is rendered out of the schema of the module to which a configuration belongs. The editor always has a title and a page header. The conent beneath the header depends on what you are currently viewing. If you jump into a subcontainer, the editor will show the content of the subcontainer. You can use the *Node Outline* view to see your current position and to navigate inside the editor.

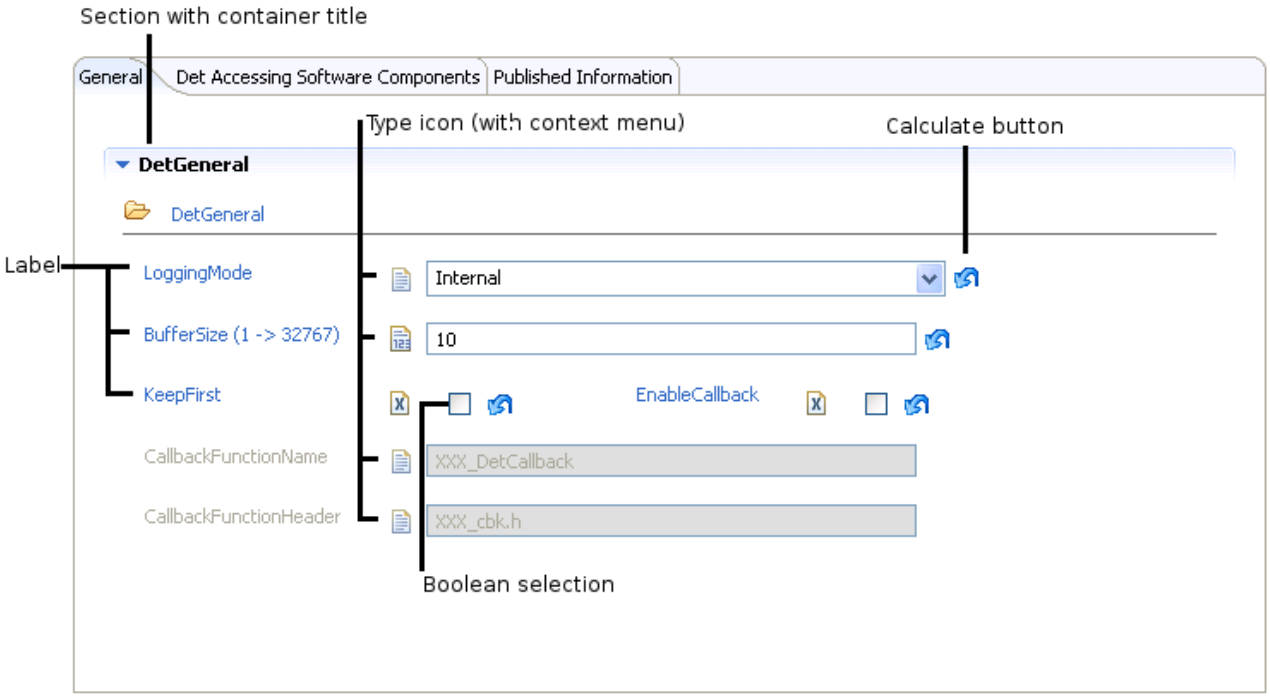
4.9.2.1. Page header, page name, and tabs



Name	Explanation
Title	Shows the title text with the module name, the module configuration name in brackets, and a close button. **Note** The equivalent of a title in an Autosar model is the name of the `CONTAINER-DEF` or `MODULE-DEF`. Hint: Place your mouse arrow over the title to display a tool tip with the module configuration name, the module ID, and the project the editor belongs to.
Editing area	Shows the actual configuration parameters. This area is organized so that it shows one page/tab at a time. Pages are organized hierarchically in form of a tree, effectively making the editor the home page and arranging all other pages below this home page.

Name	Explanation
	Note The equivalent of a page in an Autosar module is a `CONTAINER` or a `MODULE-CONFIGURATION`.
Page header	Shows the title of the page on the left.
@ @ icon	Select to enter an XPath path to a configuration parameter. The editor navigates to this parameter.
arrow up icon	Select to move one page up in the page hierarchy. The editor navigates to the page that hosts the current page.
house icon	Select to navigate to the home page of this module.
Name	Shows the name (in case of an Autosar model the `SHORT-NAME`) of the element in the configuration that is represented by the page. If the page represents a `MODULE-CONFIGURATION` element, you cannot change the name. If the page represents a `CONTAINER`, you may change the name.
Tabs	Show the parameter in the editor. Just one tab at a time is displayed. **Note** - The General tab contains all containers (Autosar: `CONTAINER`), parameters (Autosar: `*-VALUE`), and references (Autosar: `REFERENCE-VALUE`) that appear exactly once (Autosar: `LOWER-MULTIPLICITY==UPPER-MULTIPLICITY==1`). - All elements that appear more than once are displayed a tab of their own. - The Autosar `INSTANCE-REFERENCE-VALUE` is also displayed in its own tab. This is so because an instance reference contains a table with context references.

4.9.2.2. Sections and parameters



Name	Explanation
Section	Parameters and references in the General tab are sorted into sections. For Autosar, any section represents a `CONTAINER` that appears only once inside the element. No empty sections are generated.
Parameter	Parameters that occur just once (not list parameters) are displayed in the *General* tab. They are visualized differently depending on which type of parameter they are.
Label	The label describes the element. The label usually indicates the parameter name (Autosar: `SHORT-NAME`), but not necessarily so. It may also be named differently, e.g. to indicate its functionality. **Note** If the parameter contains an unsaved value, the label turns blue and the editor tab is displayed with a star.
Type icon	The type icon indicates the type of parameter and can be one of the following: to enter arbitrary text strings.

Name	Explanation
	▶ to enter integer or floating point numbers. ▶ to enter Boolean values. The following overlay icons are shown on top of the type icon: ▶ indicates a wrong parameter. ▶ indicates an editor warning. ▶ element is currently set to its default value. ▶ element value is calculated. ▶ parameter is set to a pre-configured value. ▶ parameter is set currently to a recommended configured value. ▶ element value was imported. ▶ The icon may host a context menu with further functionality, depending on the parameter. **Tip** Point your mouse cursor over the icon. If the cursor turns into a hand, a context menu is available. You can open it by right-clicking on the icon.
Entry field	The entry field next to the type icon shows a parameter value, which you may change. The type of entry field (available: text, checkbox, combobox) depends on the parameter type: ▶ Arbitrary text strings are displayed as textboxes or as comboboxes if they have a predefined value list. No context menu is provided. ▶ Boolean values are displayed as checkboxes. ▶ Integers or floating point numbers are displayed as textboxes or as comboboxes if the editor provides a predefined value list. A context menu enables selecting the display formats: ▶ Autodisplay as `HEX`

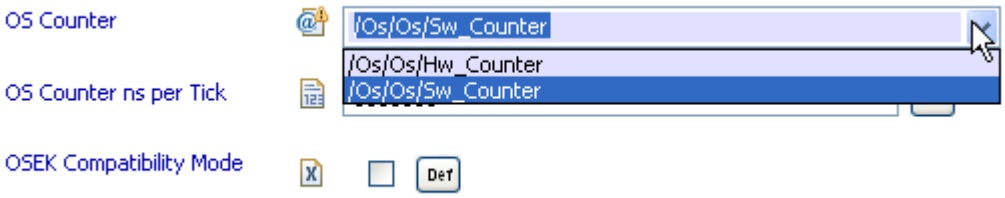
Name	Explanation
	▶ Autodisplay as `BIN` ▶ Autodisplay as `DEC` ▶ Autodisplay as `OCT` The display format does not affect the format in which the number is stored in the configuration. The format just determines the display format not the entry format to enter the number. You may select *Show separators* from the context menu to place separators (periods or spaces) into the displayed number. You may enter integers in the following number formats, depending on their prefix: ▶ hexadecimal digits: if the number starts with a `0x`, it is interpreted as a hex number; ▶ binary digits: if the number starts with a `b`, it is interpreted as binary number; ▶ octal digits: if the number starts with a `\0` it is interpreted as an octal number; ▶ other digits: all other numbers are interpreted as decimal numbers
Calculate button	Selecting the **Calculate** button initiates specified parameter calculations. The icon itself depends on the current state of the parameter. The following operations are available: ▶ The current value of the element is not edited manually but comes from another source (e.g. default value, importer). Pressing the button marks the element as having been manually edited. With this operation you can choose to accept the value. Operations like running autoconfigure wizards or importers that would normally overwrite the default, calculated or imported values do not touch the element any more. ▶ The current value of the element is not edited manually but comes from another source (e.g. default value, importer). Pressing the button marks the element as having been manually edited. With this operation you can choose to accept the value. Operations like running autoconfigure wizards or importers that would overwrite default, calculated or imported values do not touch the element any more.

Name	Explanation
	The small calculator icon indicates that the value of the element depends on the value of other elements and that you should let EB tresos Studio calculate its value. ▶ The current value of the element was edited manually. Pressing the button recalcutes the element value and marks it as a calculated default value. ▶ The current value of the element was edited manually. Pressing the button sets the value of the element to its default. ▶ The current value of the element was edited manually. Pressing the button clears the value of the element because there is no default value to set.

4.9.2.3. References

References are elements that refer to parameters. Autosar distinguishes between `REFERENCE-VALUE` and `INSTANCE-REFERENCE-VALUE`:

- `REFERENCE-VALUE` (simple reference) references a single parameter with a reference path
- `INSTANCE-REFERENCE-VALUE`s (instance reference) need a target reference path and multiple context reference paths to reference a parameter



In the entry field of a simple reference, you may enter:

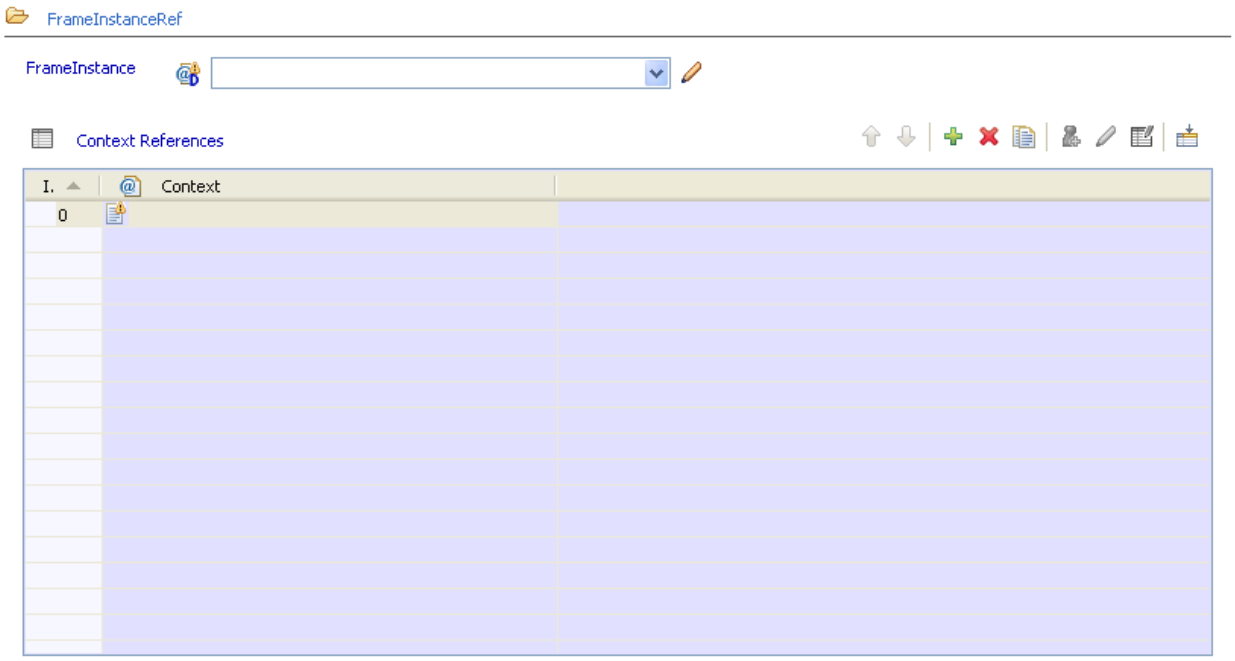
- a path expression,
- an Autosar path,
- or select an Autosar path from the paths determined in the parameter definition.

Note

If the reference element is not part of the loaded configuration and EB tresos Studio can thus not verify the path, the element is marked with a warning icon.

Tip

Navigate to the editor of the referenced parameter via the context menu of the type icon. This is possible only if the parameter is part of the configuration project.



Autosar instance references are made up of a target reference and a list of context references.

In the editor, the target reference is represented by a dropdown listbox and the context references are shown as the elements of the table. The dropdown listbox lets you select the target of the reference.

If the target cannot be uniquely referenced in your configuration, you need to specify the context of the target reference by adding one or more context references to the table. If valid selections are available in your configuration, all dropdown listboxes of the instance reference let you select them.

4.9.2.4. Choices

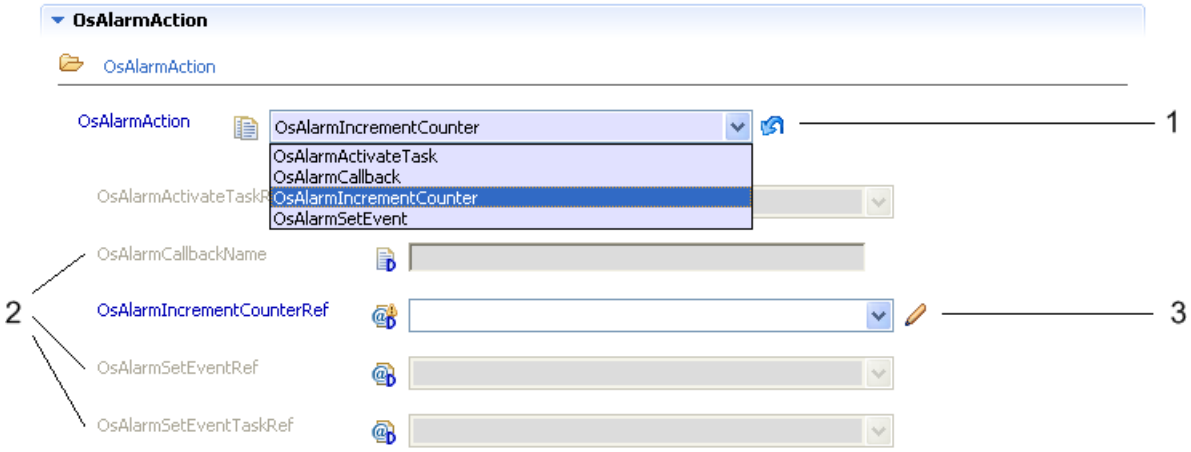
Autosar provides choices to select between different parameter sets. A choice in the Autosar parameter definition model consists of a `CHOICE-CONTAINER-DEF` that includes `PARAM-CONF-CONTAINER-DEF`s.

Tip

To configure parameters, select one of these `PARAM-CONF-CONTAINER-DEF`s and create a `CONTAINER` for it.

A choice is displayed as a section. The section contains a dropdown listbox which represents the choice itself and several subelements, which represent the parameters of the choice. These parameters correspond to the selectable values of the choice. All but one of the parameters are disabled.

- To control, which parameter is enabled, select a different value from the dropdown listbox.
- If you change the listbox, the parameter previously enabled is now disabled and the newly selected one is enabled.



Number	Explanation
1	Shows a combobox for selecting which of the subcontainers is going to be active. Only one subcontainer can be selected.
2	All unselected subcontainers are disabled.
3	The selected subconatiner is enabled.

Hint: Make sure to remember the values so that you may switch between the containers.

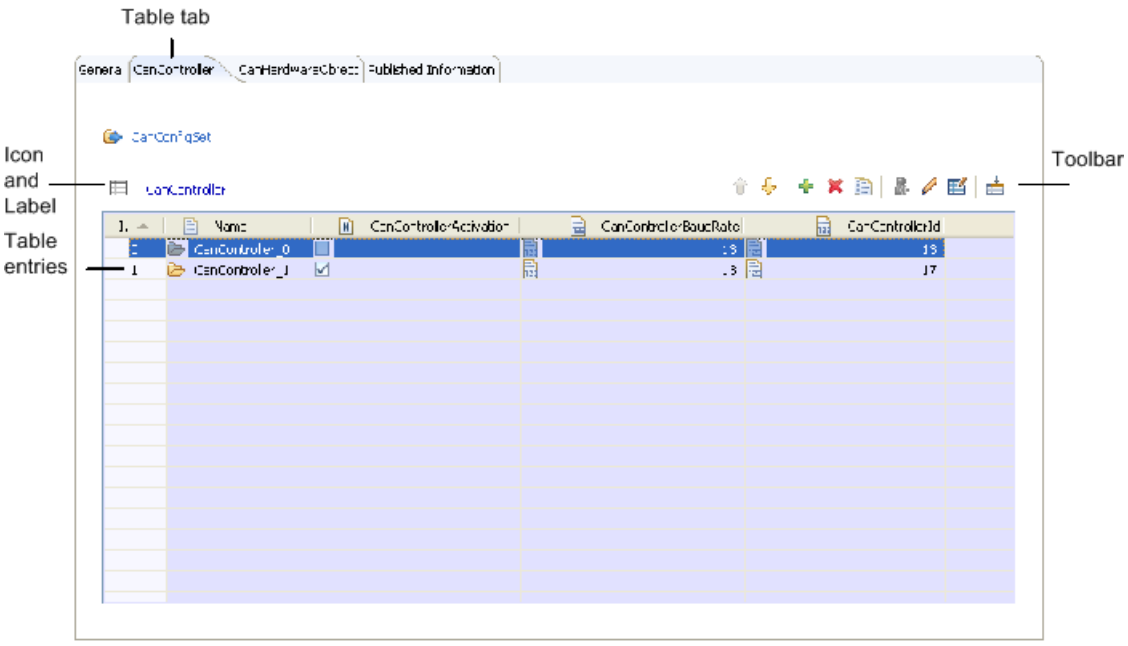
4.9.2.5. Tables

Tables represent all elements for the configuration of a user-selectable number. In Autosar terms this means, all elements that have a

- `LOWER-MULTIPLICITY != UPPER-MULTIPLICITY` , or
- `LOWER-MULTIPLICITY == UPPER-MULTIPLICITY > 1`.

It depends on the parameter and on the context you are using it in, how many instances of any one element you need.

Parameters that must be displayed in a table are provided with their own tab.



A table consists of:

- a label with an icon
- a list with table entries
- a toolbar

Name	Explanation
Label	Displays the name of the list and indicates the saved status of entries: - if the list contains unsaved changes, the label changes to blue
Icon	Shows an error or warning marker if there are too few or too many entries in the table. Hint: Click on the icon to display information about the table itself in the Element Outline view.
List with table entries	The layout of the list of table entries depends on the type of elements displayed. A table may display elements that group other elements, such as - Autosar containers are displayed with an index column and a column in which the container name is editable. Additional columns may be available for parameters beyond the container. - To navigate to the editor page where the container content can be edited, double-click a cell in the index column. The *Editor* view opens up the tab of the container content.

Name	Explanation
	▸ To display information about the selected entry, click once on a cell in the index column. The information is displayed in the *Element Outline* view. ▸ Choices are displayed at least with an index column, a name column, and a choice column. ▸ To edit the content of the choice, double-click a cell in the index column. The *Editor* view opens up the tab of the choice content. ▸ To display information about the selected entry, click once on a cell in the index column. The information is displayed in the *Element Outline* view. ▸ Parameters are displayed in two columns, the first is the index column, the second shows the actual parameter value. The icon in the column header of the second column represents the entry type (reference). ▸ References are displayed in two columns, the first is the index column, the second shows the actual reference value. The icon in the column header of the second column represents the entry type (reference).
Toolbar	Depending on the table, a toolbar with icons is available. Unavailable icons are grayed out, the available icons are colored: ▸ Select to move selected elements up in the list. ▸ Select to move selected elements down in the list. ▸ Select to add new element with default values. ▸ Select to remove selected elements. ▸ Select to duplicate selected elements. ▸ Select to add required elements. ▸ Select to open row in a separate editor or window. ▸ Select to open the *Bulk Change* editor that allows to edit multiple rows simultaniously. This button is enabled when at least one row is selected. ▸ Select to edit visibility of columns.

To help increase navigation and editing speed, tables provide keyboard shortcuts:

Shortcut	Explanation
Arrow keys	The arrow keys allow to change the row currently selected.
Control-A	Selects all rows in the table.
Delete	Deletes the rows selected.
Insert	Inserts a new row at the end of the table.
F2	Starts editing the first editable column in the row currently selected.
▶ **Tab** ▶ **Shift-Tab**	Allows to switch between edited cells when editing table entries.
Space	Enters the editor for the container in the row currently selected.

Tip

To add a new entry to the table, double-click in the unoccupied part of the table. A new entry is automatically inserted at the end of the list.

4.9.2.6. Optional elements

An Autosar configuration contains optional configuration elements. You may decide if you want these configuration elements to be part of the configuration or not.

- To add the element to the configuration select this toggle button when it is brown.

  The button's color turns green. The element is added to the configuration.

  CanIfUserDispatchHeaderFile

- To delete the element from the configuration select this toggle button when it is green.

  The button's color turns brown. The element is no longer part of the configuration. But the configured values of the element (and its child elements, if any) are saved. You may add it again at anytime, it is not permanently deleted.

  CanIfUserDispatchHeaderFile

If an element is switched off (not part of the configuration) the GUI of the element is greyed out.

4.9.2.7. Changing multiple rows in a table

To change several rows of a table at the same time, EB tresos Studio provides the *Bulk Change* editor. By automating the value change over several rows, this *Bulk Change* editor keeps you from having to change the same value in different rows manually over and over again.

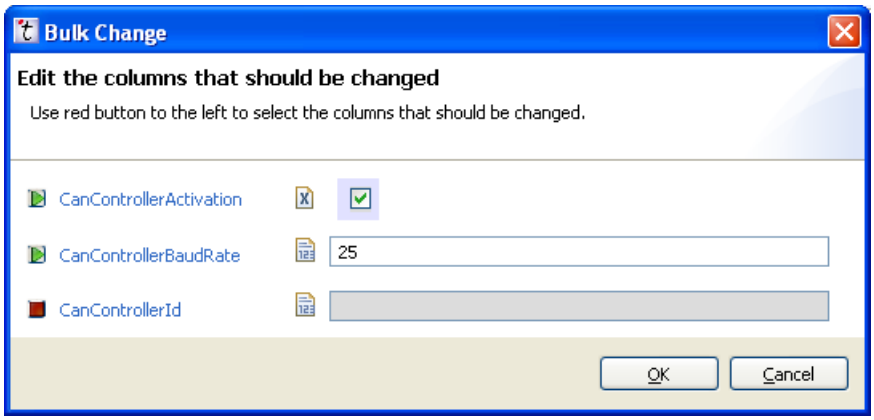
- To bulk change values in several table rows, select all table rows in which you would like to change values. For details, please refer to Section 4.9.2.5: Tables.
- Now open the *Bulk Change* editor.

  To open the *Bulk Change* editor, press the button in the table toolbar.

  The *Bulk Change* editor opens up.
- Alternatively, you may open the *Bulk Change* editor from the context menu of a table.

  To open the *Bulk Change* editor via the context menu of the table, right-click on the table, then select *Bulk Change...* from the context menu.

  The *Bulk Change* editor opens up.



The *Bulk Change* editor provides an entry field for each column currently shown in the table.

- To change which columns are displayed, use the toolbar of the table. Instructions are available at Section 4.9.2.5: Tables.
- To edit the value of the field, press the red button to the left of each respective entry field. You thus enable the respective entry field.

All columns, for which the entry field is enabled, are changed to the value entered when you close the *Bulk Change* editor.

- To execute the change, press **Ok**.
- To cancel the operation, press **Cancel**.
- To undo the operation, press the undo button in the EB tresos Studio toolbar.

4.9.3. Autoconfigure wizards

When you are using the configuration editor you have to edit all values manually. This can sometimes be a very tedious task. EB tresos Studio supports you with those routine jobs by supplying so called *autoconfigure*

wizards. These wizards can cover all kinds of tasks. Their main goal is to guide you through difficult configuration jobs and to handle routine tasks that require no user-interaction.

Each wizard comes with its own graphical user interface which is specially suited for the wizard's task. All wizards have in common that you must configure them before you can run them on your project.

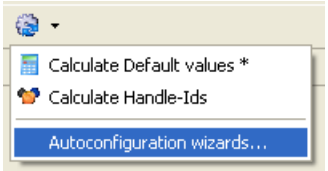
The list of available wizards can be extended by the installed modules. By default EB tresos Studio ships with two wizards: the *calculate default values* wizard and the *handle id* wizard. These two wizards are decribed in the following section. If modules make contributions to the available wizards, you should find documentation about those wizards in the module's documentation.

4.9.3.1. Configuring Autoconfigure wizards

Before autoconfigure wizards can be used, they have to be configured.

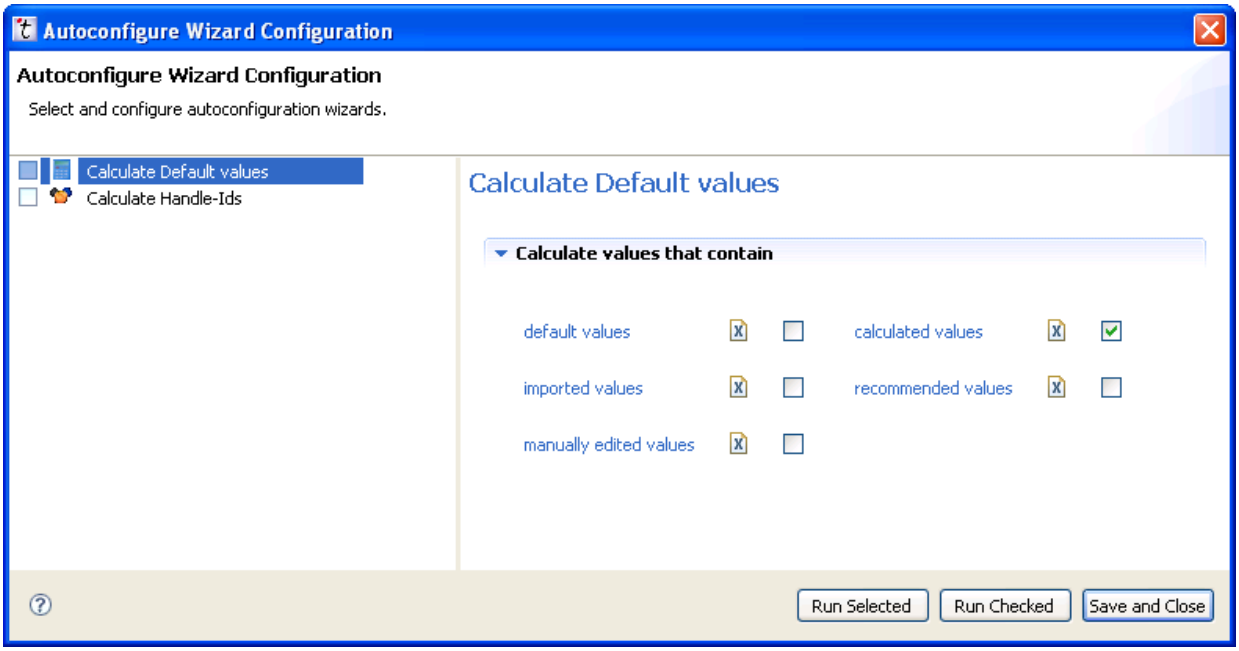
- To configure an autoconfigure wizard, select the project on which you would like the wizard to operate from the *Project Explorer*.
- In the toolbar, select the arrow next to the *Run all enabled autoconfigure wizards* button.

  A submenu opens up.

- Select *Autoconfiguration wizards*.

  The *Autoconfigure Wizard Configuration* dialog opens up.



On the left side, the window lists all *Autoconfigure* wizards. Select the wizard you want to configure from the list. The right part of the dialog window now shows the user interface of the wizard. The exact look depends on the wizard you have selected. There may be warnings or errors shown at the top of the window if the wizard is not configured correctly.

After you are finished confguring the wizard, you can close it by clicking on **Save and Close**.

4.9.3.2. Running Autoconfigure wizards

You can choose if you want to run a single wizard, or if you want to run several wizards in a row. In any case there is a result window shown after the wizard has finished the work. This window gives you information about the outcome of the wizard.

Additionally, after configuration, some autoconfigure wizards can also be run from the commandline. A general explanation on how autoconfigure wizard can be launched from the commandline can be found in Section 4.12.2.5: Executing autoconfigure wizards on projects. Please consult the documentation of the autoconfigure wizard you want to run to get additional information like commandline options or prerequisites.

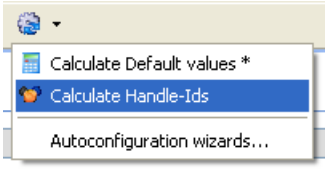
4.9.3.2.1. Running a single autoconfigure wizard

There are two ways to run a single autoconfigure wizard:

Alternative 1:

- select the project on which you would like the wizard to operate from the *Project Explorer*.
- In the toolbar, select the arrow next to the *Run all enabled autoconfigure wizards* button.

A submenu opens up.



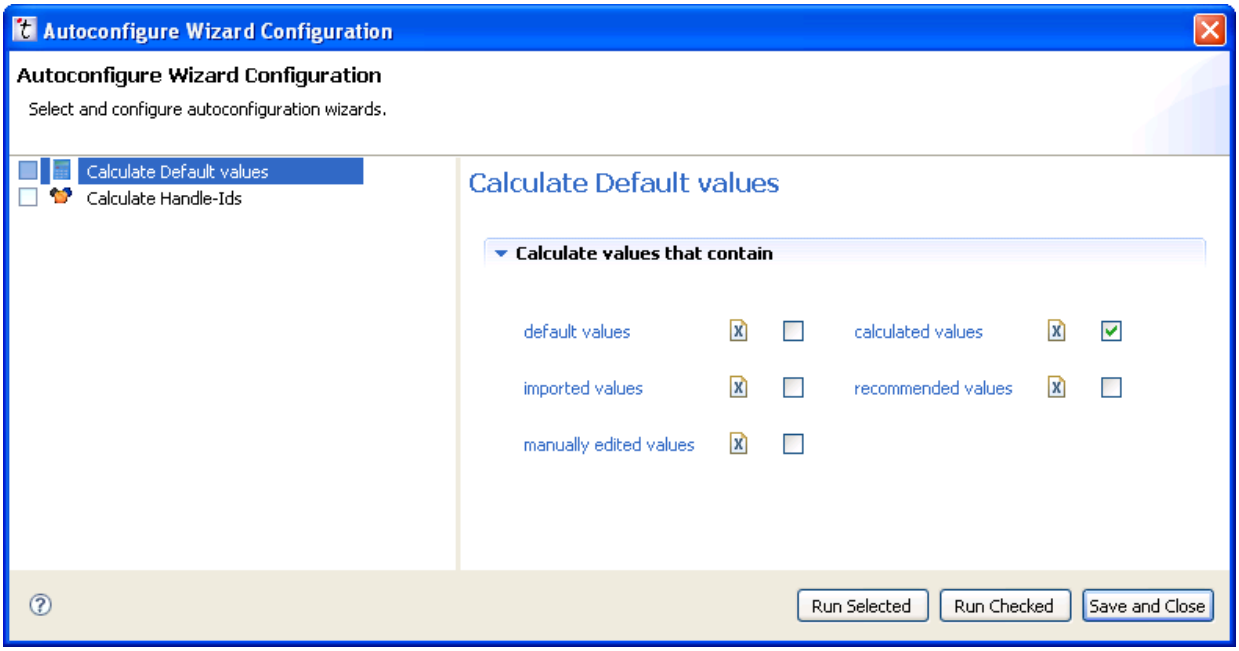
- In the submenu, click on the wizard you want to run. After the wizard has finished, you see the result dialog (see Section 4.9.3.2.3: Result feedback).

Alternative 2:

- select the project on which you would like the wizard to operate from the *Project Explorer*.
- In the toolbar, select the arrow next to the *Run all enabled autoconfigure wizards* button.

  A submenu opens up.
- In the submenu, select *Autoconfiguration wizards*.

  The *Autoconfigure Wizard Configuration* dialog opens up.



- In the dialog, highlight the wizard you want to run by selecting it from the list on the left side.
- The configuration for the wizard is shown on the right side of the dialog. If you want to you can make changes to the configuration before running it.
- Click on the **Run Selected** button to run the selected wizard.

Tip

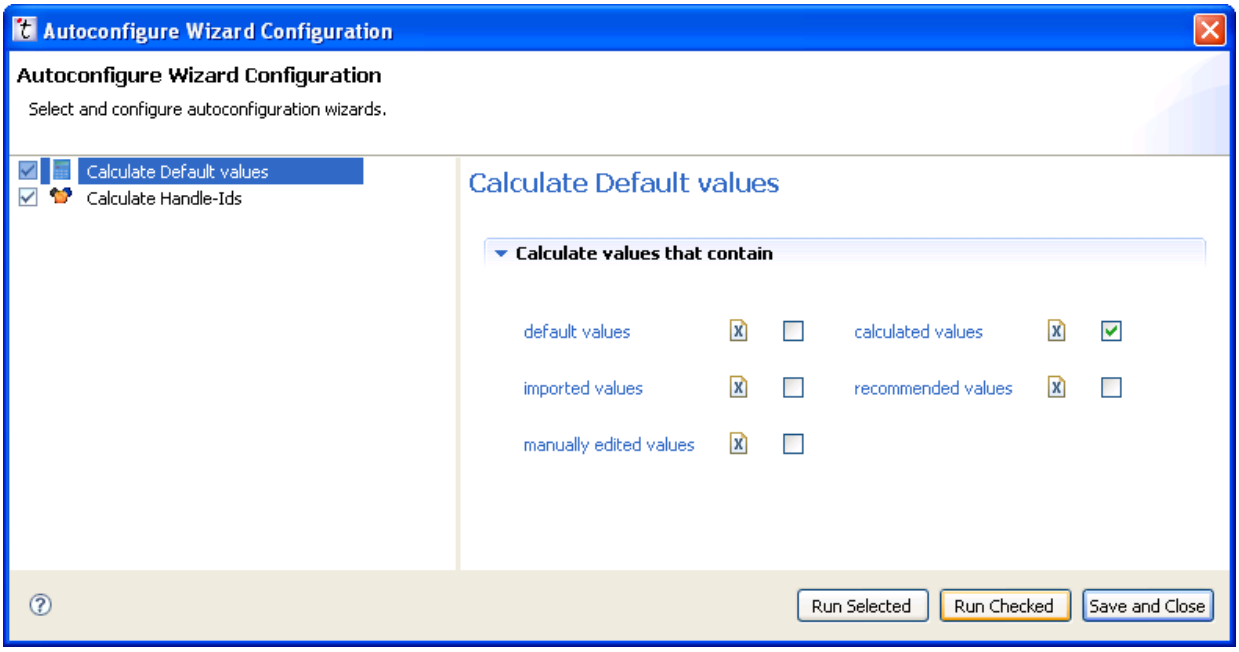
In the image above, clicking the **Run Selected** button would run the *Calculate Default values* wizard.

After the wizard has finished, you see the result dialog (see Section 4.9.3.2.3: Result feedback).

4.9.3.2.2. Running multiple autoconfigure wizards

You can define a set of autoconfigure wizards which then can be run directly after another with only one command. This is also called *batched execution*. To define this set of wizards:

- Open the *Autoconfigure Wizard Configuration* dialog.



- On the left side of the dialog you see the list of autoconfigure wizards. For each wizard that you want to include in the batched execution, check the checkbox that is in front of the wizard's label.

Note

Before you can check a wizard you possibly need to configure it first.

There are two ways to run the wizards you have just checked in the dialog.

Alternative 1:

- In the *Autoconfigure Wizard Configuration* dialog, click on the **Run Checked** button.

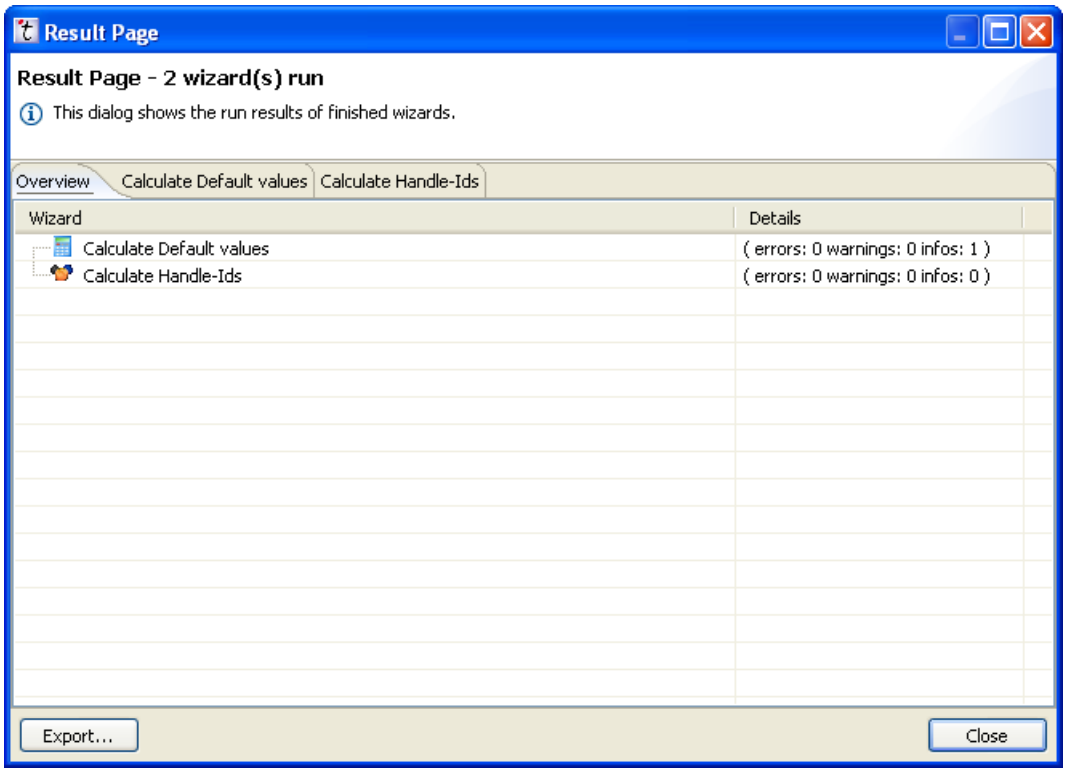
- After all wizards have finished, you see the result dialog (see Section 4.9.3.2.3: Result feedback).

Alternative 2:

- Close the dialog by clicking on the **Save and Close** button. This will save the state of the checkboxes.
- Make sure the configuration project on which you want the wizards to run is still selected in the *Project Explorer*.
- In the toolbar, click on the **Run all enabled autoconfigure wizards** button. All wizards that you have checked earlier are run.
- After the wizards have finished, you see the result dialog (see Section 4.9.3.2.3: Result feedback).

4.9.3.2.3. Result feedback

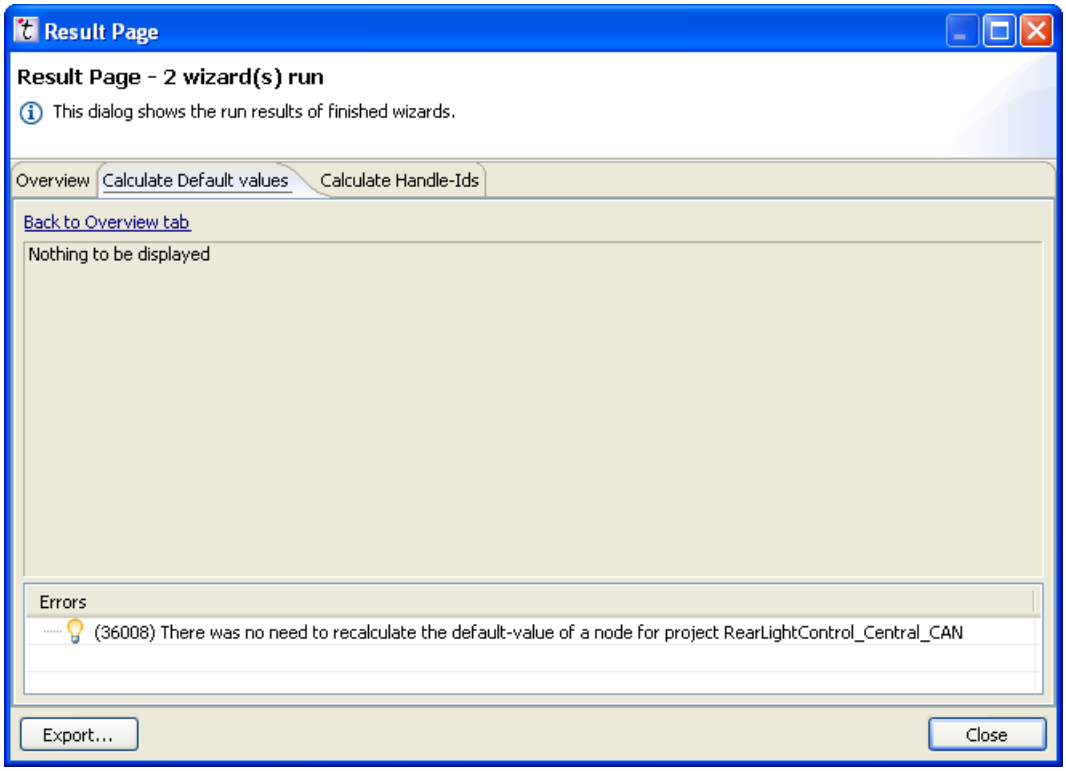
After one or more autoconfigure wizards have completed exectuion, the *Result Page* dialog is shown.



The *Result Page* dialog displays to result of each wizard run. The dialog consists of several tabs. The exact amount of tabs and their content depends on the wizards that have been executed.

The leftmost tab always is the *Overview*. This tab shows you a summary of the results of each wizard. In the image above, you see the results of two wizards, the *Calculate Default values* and the *Calculate Handle-Ids* wizard. The *Calculate Default values* wizard had one *info* message as output. To view this message, you need to click on the *Calculate Default values* tab.

Every wizard that was executed has a result page in the dialog. The page is accessible by clicking onto the tab with the same name as the wizard.



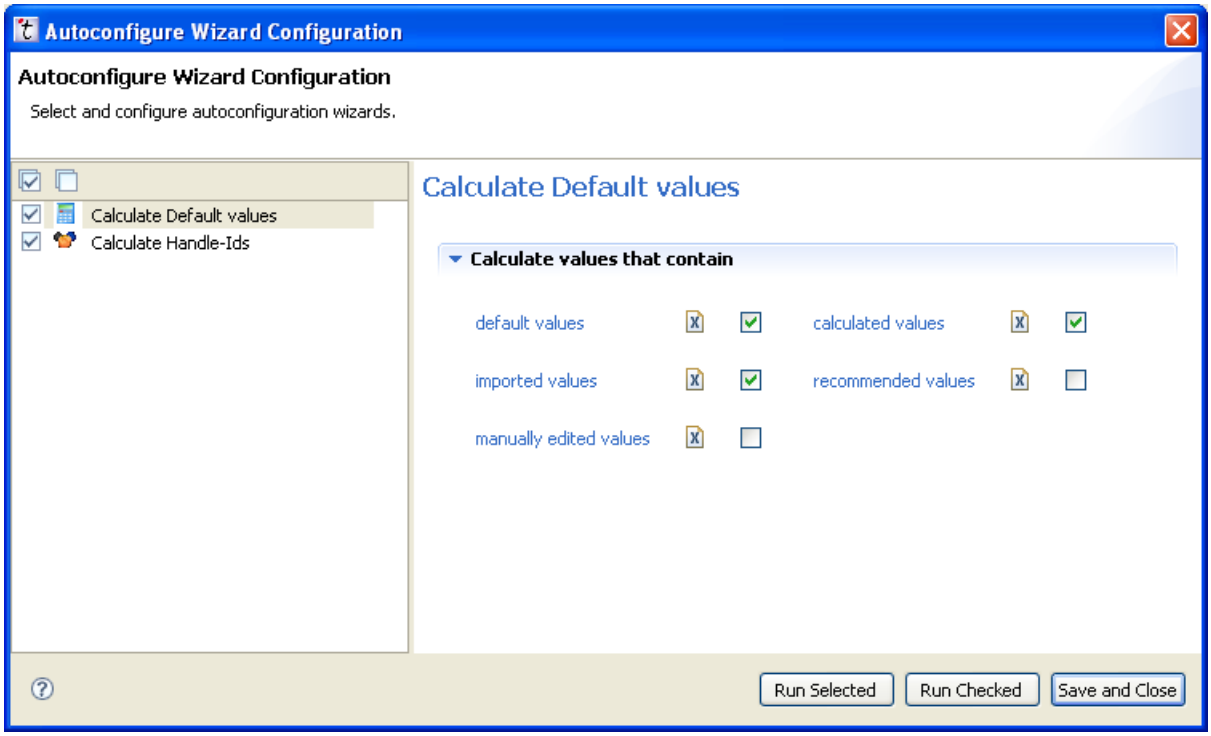
If the wizard has any output, there is a list visible at the bottom of the window, labeled *Errors*. This list contains all errors, warnings and info message that occurred while running the wizard. If a wizard wants to display more detailed information, it can fill the area in the middle of the dialog with more GUI elements. Not all wizards use this feature.

In the image above, the wizard does not make use of displaying additional GUI elements, but you can see an info message at the bottom of the window.

The results of a wizard can be exported to a file by clicking the **Export** button. A dialog opens which lets you choose a filename under which the results are saved.

4.9.3.3. Configuring the *Calculate Default Values* wizard

The *Calculate Default Values* wizard recalculates the value of nodes, which must have a default value. If you chosse this wizard, the **Error Log** view is going to display a list of all values that have been recalculated/changed.

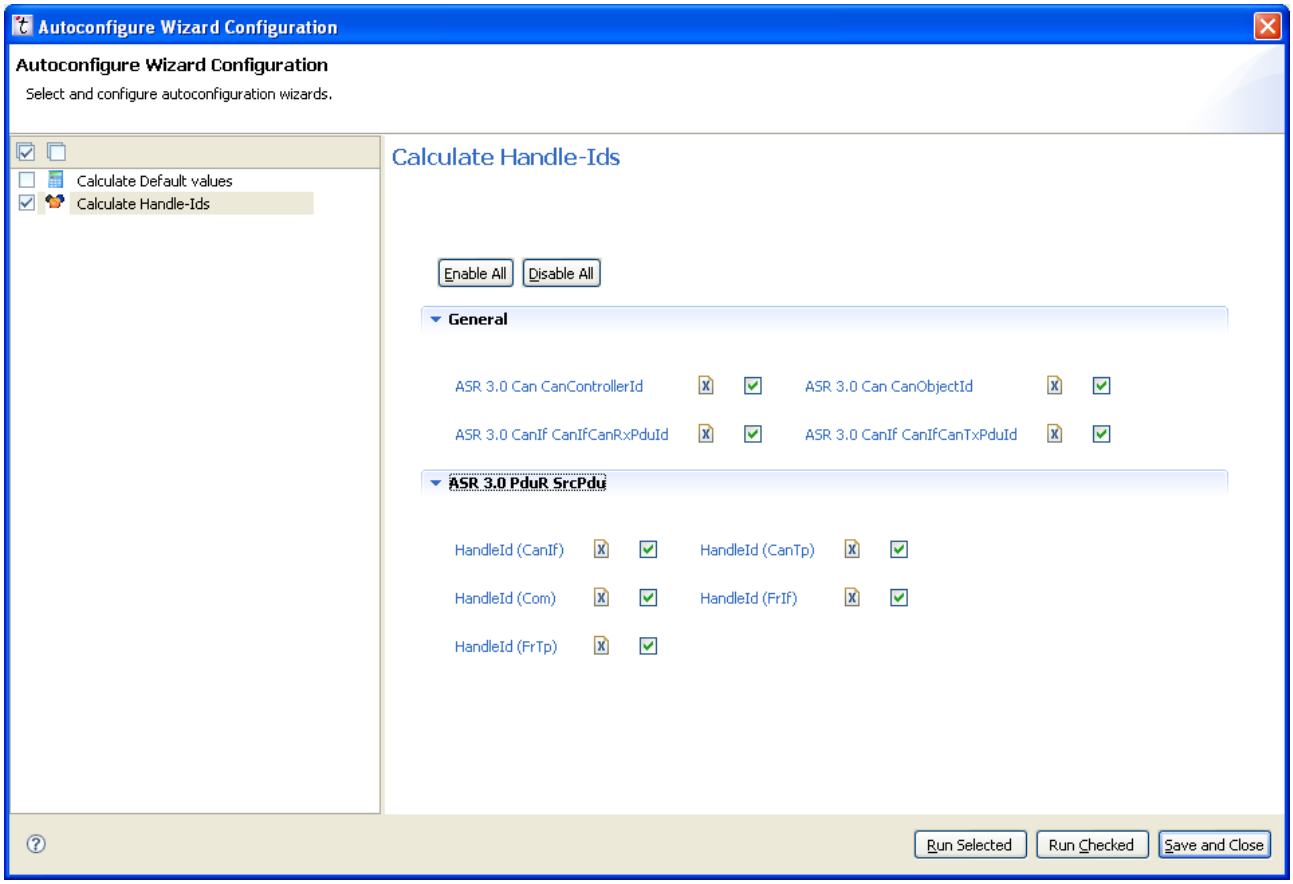


You can select which values you would like this wizard to recalculate:

- Select **default values** to recalculate all values for which the module provider defined a default value calculation formula.
- Select **calculated values** to recalculate all values which are calculated. This are, for example, the values calculated by another *Autoconfigure* wizard, e.g. handle-ID.
- Select **imported values** to recalculate all values that have been overwritten by an import.
- Select **recommended values** to recalculate all values which have been set to the recommended value.
- Select **manually edited values** to recalculate all values which you have changed.

4.9.3.4. Configuring the *Calculate Handle-Ids* wizard

The *Calculate Handle-Ids* wizard calculates the handle-IDs for the modules of the Communication stack.



The appearance of this wizard depends on the modules available in your project. There is a checkbox available for each handle id parameter. If you check a checkbox, the handle ids will be calculated for the parameter when the wizard is run.

4.9.4. Searching for configuration parameter names and their values

EB tresos Studio provides the option of searching for configuration parameter names and their values within files or within the data model.

To start the search:

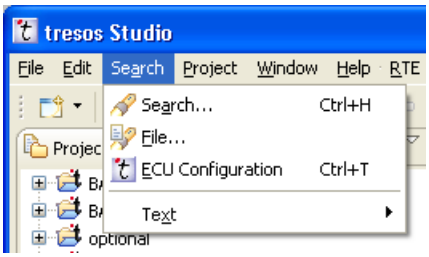
Alternative 1:

- Press **Ctrl-T**
- The *Search* dialog opens up and the *Tresos Search* tab is selected.

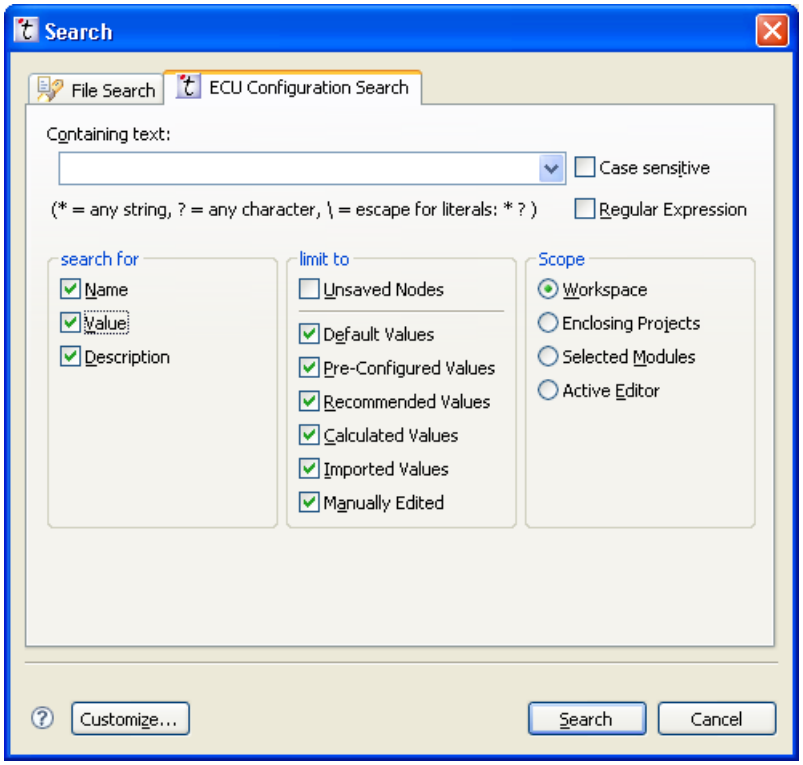
Alternative 2:

- Select the *Search* menu from the main menu.

A sub-menu opens up.



- Select the *Tresos... Ctrl+T* submenu.
- The *Search* dialog opens up and the *Tresos Search* tab is selected.

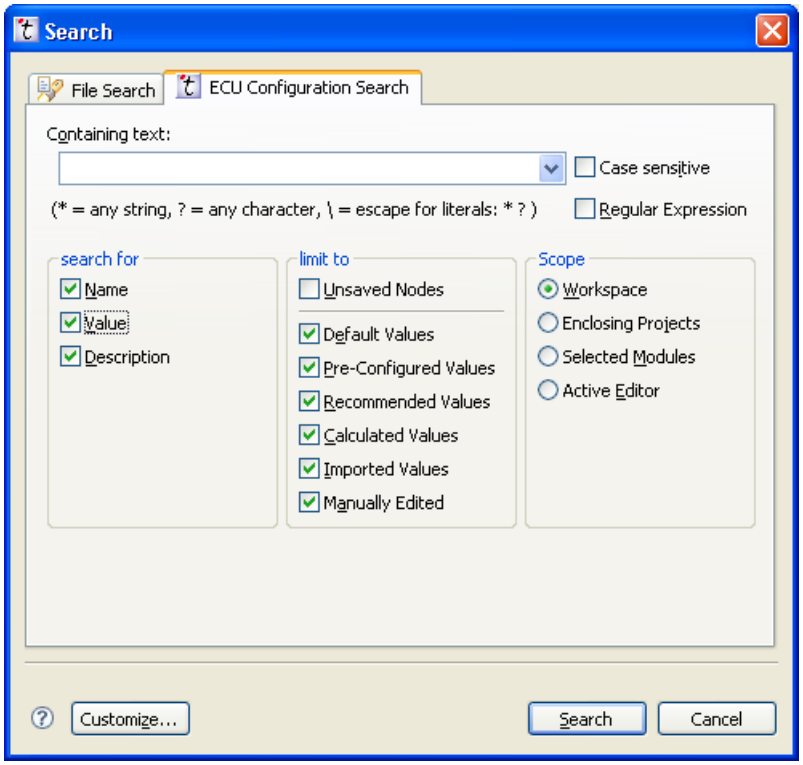


Note

The *Search* window also serves other purposes than to search within the data model of a project. Thus, it provides several other tabs. These tabs are contributed by the Eclipse framework and have no use in EB tresos Studio. Please consult the Eclipse documentation for more information.

4.9.4.1. Searching in the data model

To find configuration parameter names and their values within the data model, select the tab *Tresos Search*. As you open the tab, the last search executed is automatically restored .



- To restore an earlier search, select one of your last searches from the *Containing text* drop-down listbox.

  All options for the respective search are restored.
- To search for a new text string, select *Containing text* and type in your text.

  By default you may use the wildcards *?* for characters and * for strings. The characters ? and * can be escaped by using the \, e.g. *. If you need more advanced wildcard support, you should use regular expressions (see below).
- To search for case sensitive text strings, select the **Case sensitive** checkbox. This option is not available for regular expressions.
- To have the searched text interpreted as regular expression, select the *Regex* checkbox. Note, that this will start a content assist function.

To control where the text will be searched:

- To search for the name of nodes, select the *Name* checkbox.
- To search for the value of nodes, select the *Value* checkbox.
- To search for node descriptions, i.e. the text you see in the *Element Description (Elt. Desc.)* view on the bottom right corner of the main window, select the *Description* checkbox.

To limit, what kind parameters will be considered for the search:

- To restrict your search to unsaved nodes, i.e. those which have not yet been saved after an import, select the *Unsaved Nodes* checkbox.
- To display all elements that are set to their default value, select the *Default Values* checkbox.
- To display all elements that are set to pre-configured values, select the *Pre-Configured Values* checkbox.
- To display all elements that are set to recommended values, select the *Recommended Values* checkbox.
- To display all elements that are calculated, select the *Calculated Values* checkbox.
- To display all elements that are imported, select the *Imported Values* checkbox.
- To display all elements that are manually edited, select the *Manual Edited* checkbox.

To define the scope of the search:

- To search in all open projects, select the **Worspace** radio button.
- To search in all projects whose tree contains a selection within the *Project Explorer* view.

> **Note**
>
> The project of the currently opened editor is not automatically included.

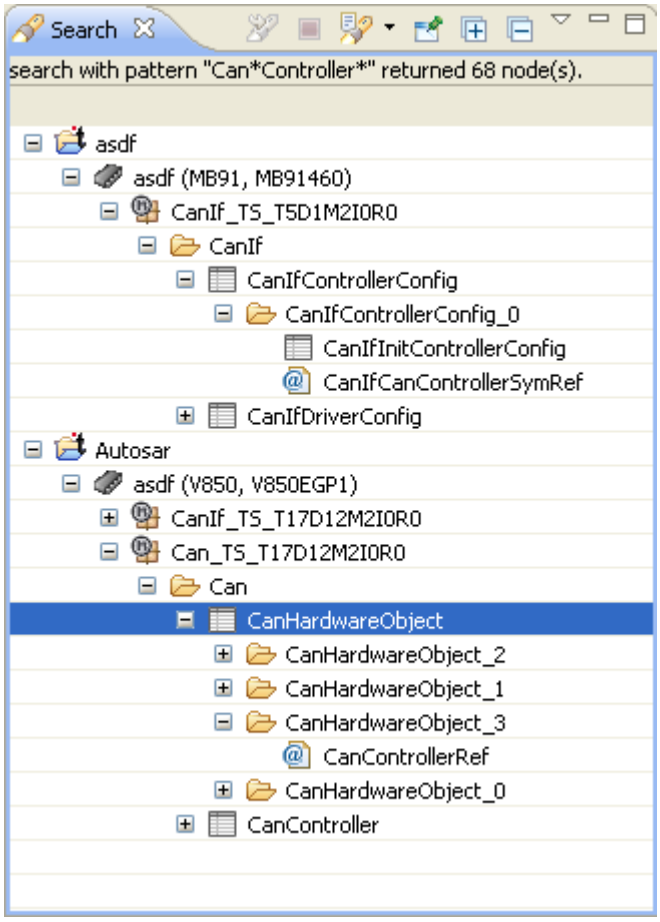
- To search for all modules whose parent- or sub-tree contains a selection within the *Project Explorer* , select the *Selected Modules* radio button.
- To search within the module of the currently active editor, select the *Active Editor* radio button.

Proceed to Section 4.9.4.2: Displaying search results to start the search.

4.9.4.2. Displaying search results

- Select **Search** to start the search.

  The *Search* view opens up on the bottom of the GUI beside the *Error Log* and the *Problems View*. The *Search* view contains all the results of the search, ie. the configuration nodes that contain the searched text.



The view displays a separate node tree for each project for which a configuration node was found. The upper part of the tree is the same as shown within the *Project Explorer* view. The lower part of the tree is the same as shown within the *Outline* view. Using the **+** and **-** icons at the top of the view expands or collapses the trees.

To open the *Editor* view of a node:

- Right-click on the node.

  A context menu opens up.
- Select *Show in Editor*.

  The configuration node is selected and displayed in the *Editor* view.

To open the *Editor* view of a leaf node:

- Double-click on the node.

  A context menu opens up.
- Select *Show in Editor*.

  The configuration node is selected and displayed in the *Editor* view.

4.10. Generating code for projects

After you have finished configuring a project you can generate the sourcecode of it. The resulting code depends on the values you have entered in the editor. A module generates code, if it registers a code generator. You can only generate code if the configuration project contains no errors.

4.10.1. Verifying a project

Some problems with the configured values of a project are only discovered at generation time. Before you actually generate the code you may choose to verify the code first. During verification the same processing steps are executed as during code generation. But the difference is that no files are written to disk. Verifying a project is only possible, if the *Problems* view shows no errors. The result of the verification is written to the *Error Log* view.

Tip

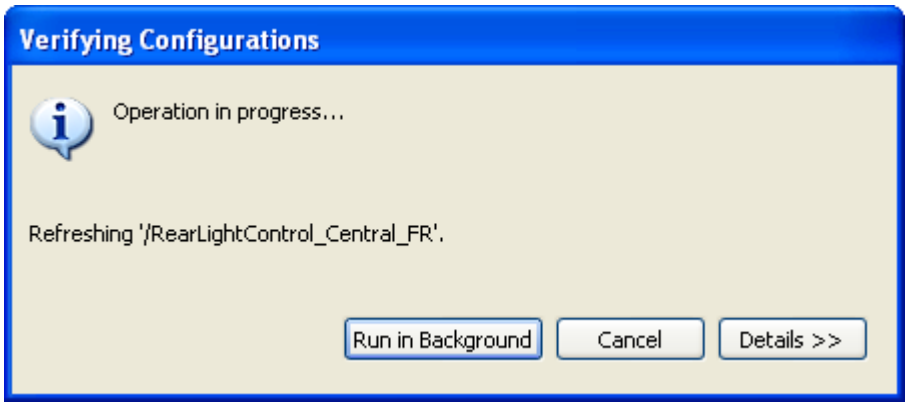
When verifying code, make sure that info messages are not blocked in the Error Log. Otherwise you might miss feedback.

There are two alternatives to verify a project:

Alternative 1: To start the generators to verify a project:

- Select an ECU configuration node within the project from the *Project Explorer*.
- Right-click on the ECU configuration node.
- Select *Verify Project*.

  A dialog window opens up and starts the verification process. The window updates while the verification is running and closes automatically when it is finished.



Alternative 2: To verify a project:

- Select an ECU configuration node within the project from the *Project Explorer*.
- Select the **Verify** button from the toolbar.

  A dialog window opens up and the tool starts verifying the project. The window updates while the verification is running and closes automatically when it is finished.

4.10.2. Generating a project

the code generation starts the code generators of all modules for which you have enabled code generation in the *Module Configuration* Dialog window (see Section 4.7.5: Disabling code generation of a module configuration). Generating code is only possible, if the Problems view shows no errors.

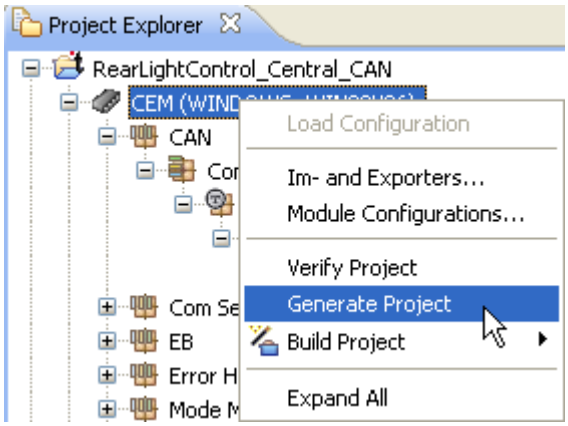
The result of the generation is written to the *Error Log* view. When generating code, make sure that info messages are not blocked in the Error Log (see Section 4.4.5.2: Filtering Error Log messages by their severity level). Otherwise you might miss feedback.

There are two alternatives to generate code:

Alternative 1: To generate code:

- Select an ECU configuration node within the project.
- Right-click on the ECU configuration node.
- Select *Generate Project*.

  A dialog window opens up and starts generating code.



Alternative 2: Alternatively, to generate code:

- Select the **Generate** button from the toolbar.

  A dialog window opens up and starts generating code.

4.10.3. Building a project

To build a project means running the registered code generator engines in a special mode, other than generate or verify. The available modes depend on the project you have currently selected and on the modules available in this project. Building code is only possible, if the *Problems* view shows no errors. The result of the build is written to the *Error Log* view .

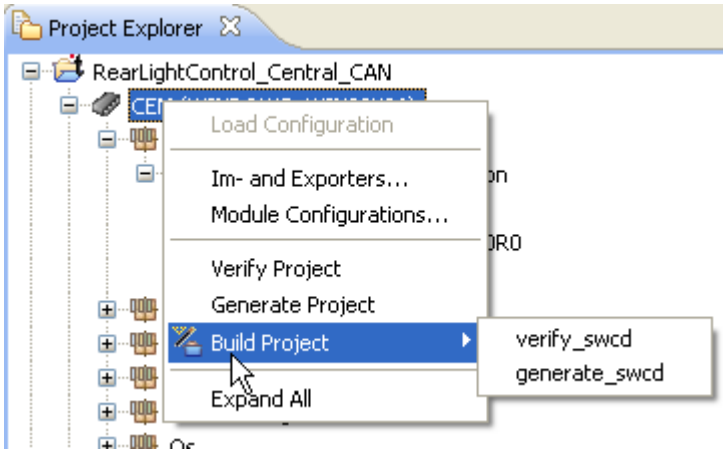
When building code, make sure that info messages are not blocked in the Error Log. Otherwise you might miss feedback.

There are two alternatives to building a project:

Alternative 1: To build code:

- Select an ECU configuration node within the project.
- Right-click on the ECU configuration node.
- Select *Build Project*.

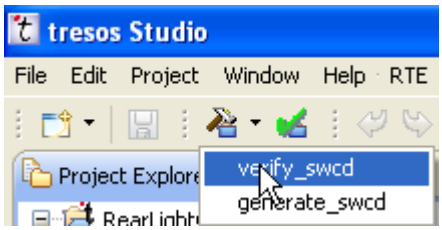
  A dialog window opens up and starts building code.



Alternative 2: Alternatively, to build code:

- Select the submenu to the **Generate** button from the toolbar (marked by a tiny downward pointing arrow).

  A context menu opens up. The submenus of this context menu depend on you project and the modules available in this project.



- Select the mode you wish to run.

A dialog window opens up and starts the code building process.

4.11. Importing and exporting

This section presents the different ways of importing and exporting projects and configuration data.

4.11.1. Exchanging a complete EB tresos Studio project with other installations

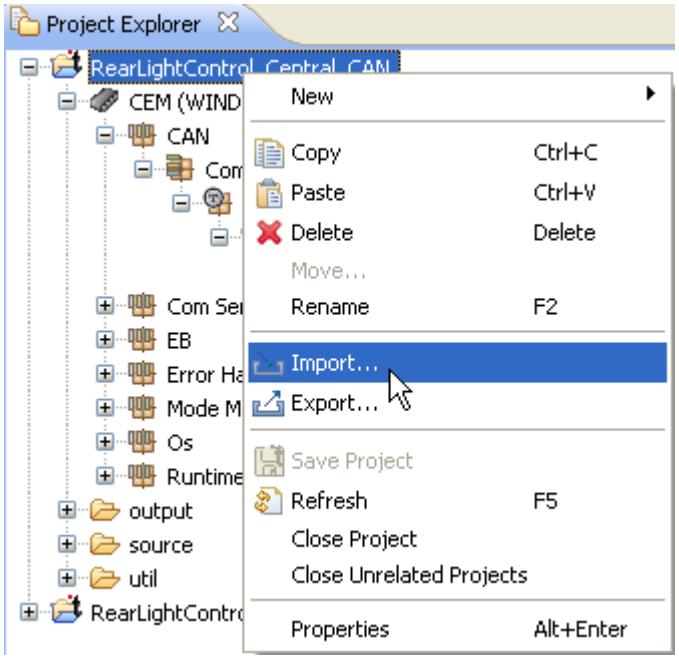
With EB tresos Studio you may import and export projects, e.g. to exchange a project with another EB tresos Studio installation, or to backup a workspace on external memory space. This feature is not the same as importing and exporting a configuration! To import or export a project configuration, please proceed to Section 4.11.2: Importing and exporting configuration data. The project import and export is done through the Eclipse framework of EB tresos Studio

4.11.1.1. Importing a project

To import a project from other locations to the currently active workspace:

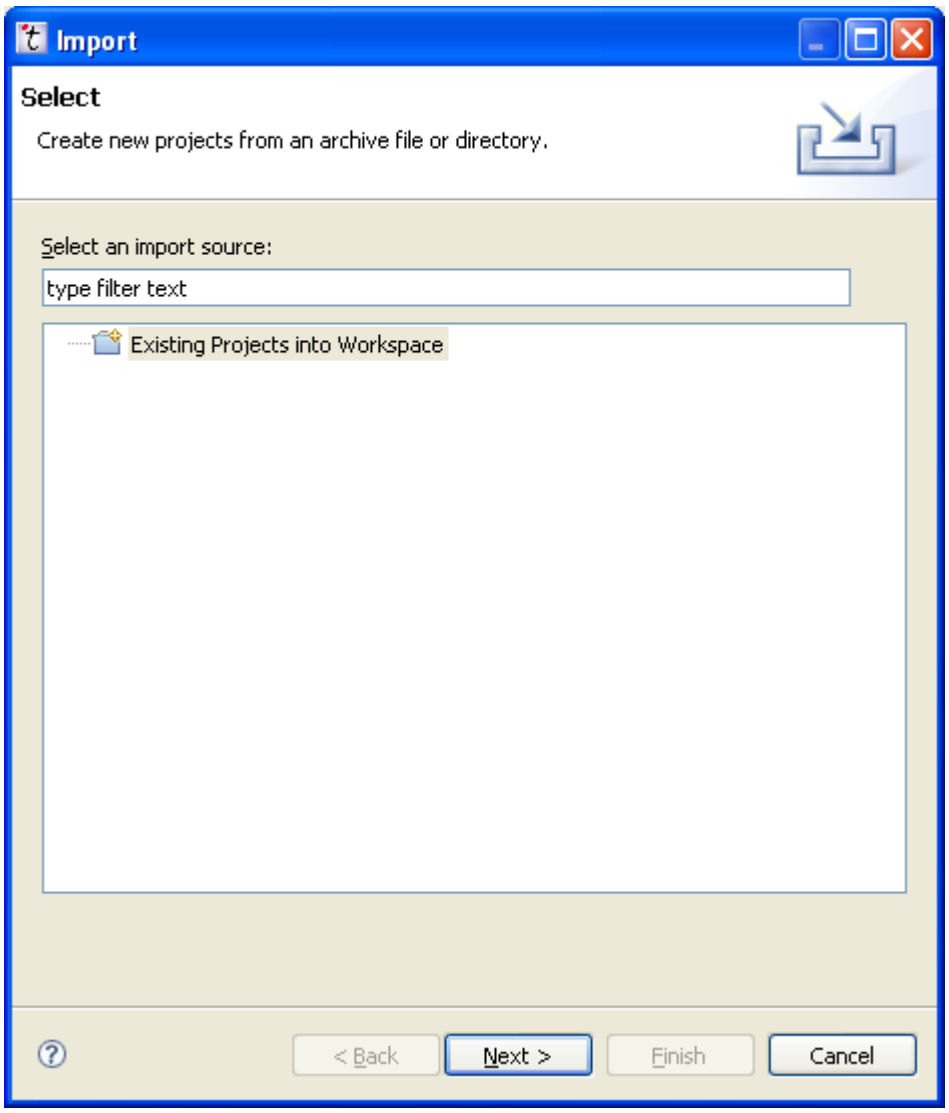
- Select the project node in the Project Explorer.
- Right-click on the node.

  A context menu opens up.



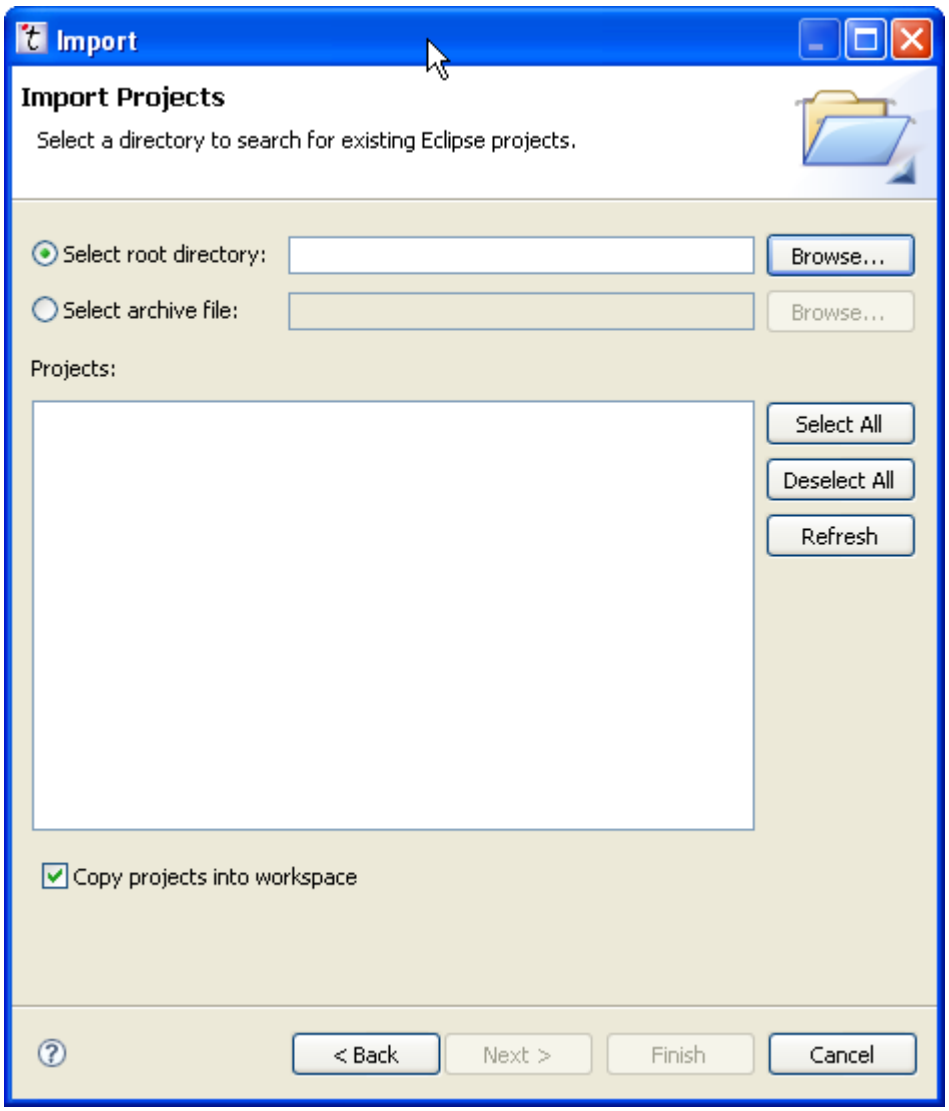
► Select **Import**.

The *Import* window opens up.



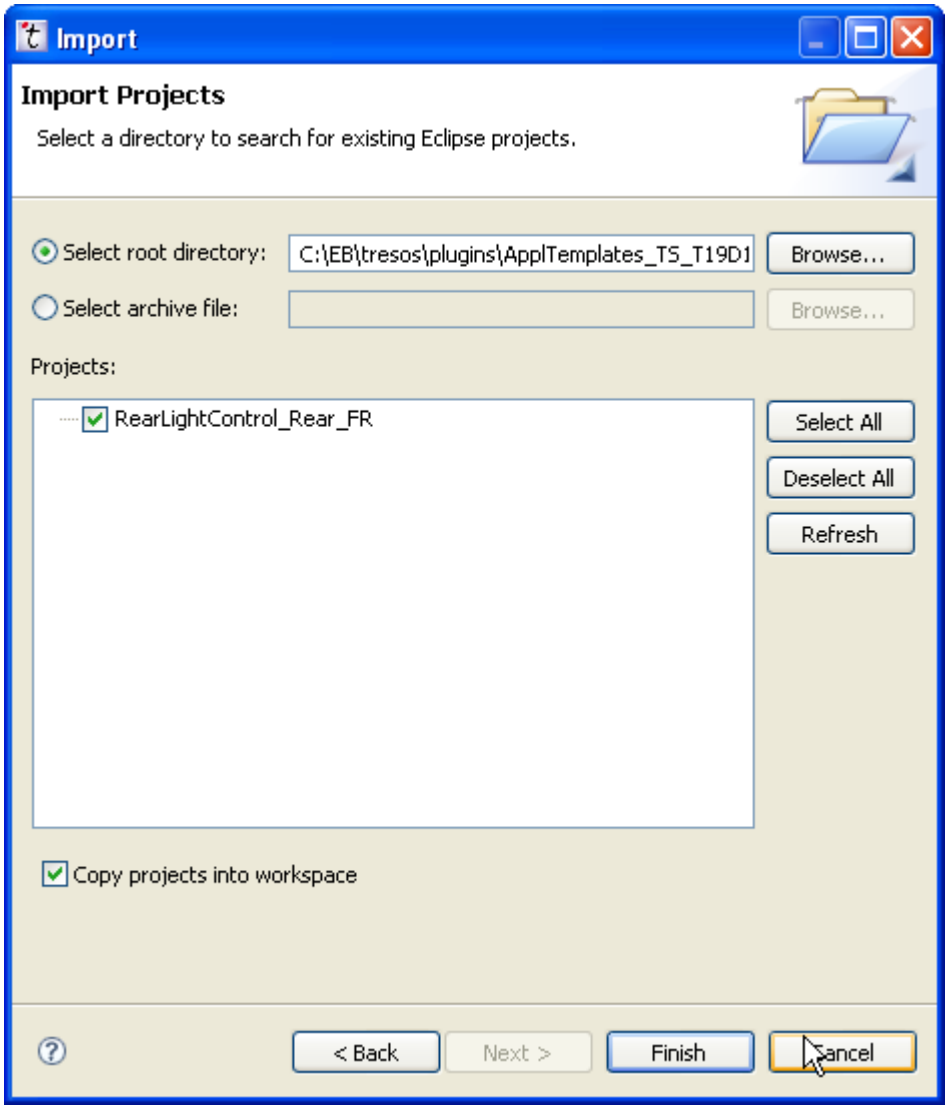
- Select *Existing Projects into Workspace*.
- Double-click *Existing Projects into Workspace*.

  The *Import* window opens up.



- If you want to import a project that is located in the file system, click onto the radio button *Select root directory* and enter the path to the project's directory in the textbox. If you click the **Browse** button, you can choose an existing directory.
- If you want to import a project that is contained in an archive, click onto the radio button *Select archive file* and enter the path and name of the archive file in which the project is contained into the textbox. If you click the **Browse** button, you can choose an existing archive file.

All projects that can be imported from your source directory or archive are now listed in the *Projects* listbox. Note that a project is not listed if a project with the same name already exists in the currently active workspace.

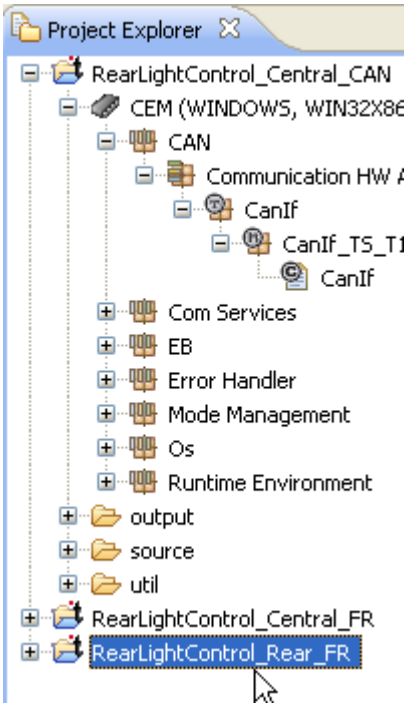


- Uncheck the projects you do not want to import from the list.
- To copy the project content into the currently active workspace, check the **Copy projects into workspace** checkbox. This option is only available if the import source is a directory and not an archive.

> **Warning**
>
> If you do not check this box, EB tresos Studio generates a linked project. This means that you will later work on the files directly inside the folder from which you imported the project and not on a local copy inside your workspace.

- After you have chosen all projects that you want to import, click the **Finish** button to start the actual import.

  The projects you imported are now listed as projects in the *Project Explorer*.

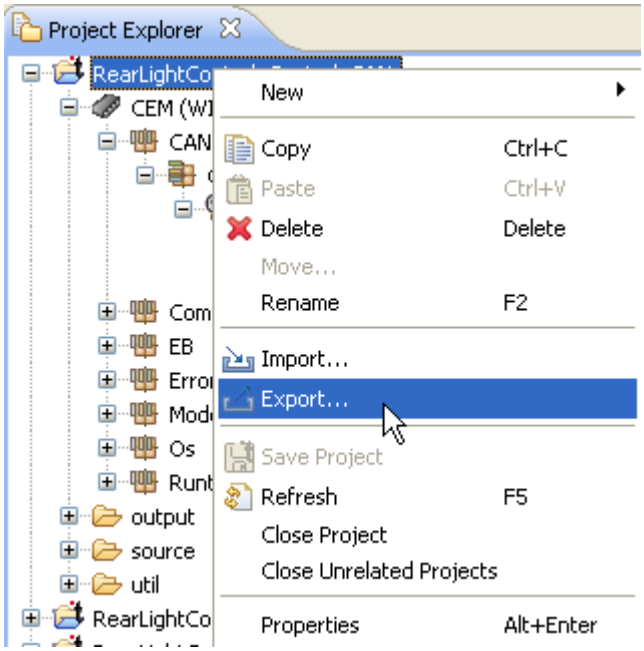


4.11.1.2. Exporting a project

If you want to share your project with other EB tresos Studio installations, or if you want to make a backup copy of a project, you can export the project. To export projects from the currently active workspace to other locations:

- Select the project node in the Project Explorer.
- Right-click on the node.

  A context menu opens up.



- Select *Export*.

  The *Export* window opens up.

Proceed with either of the following steps:

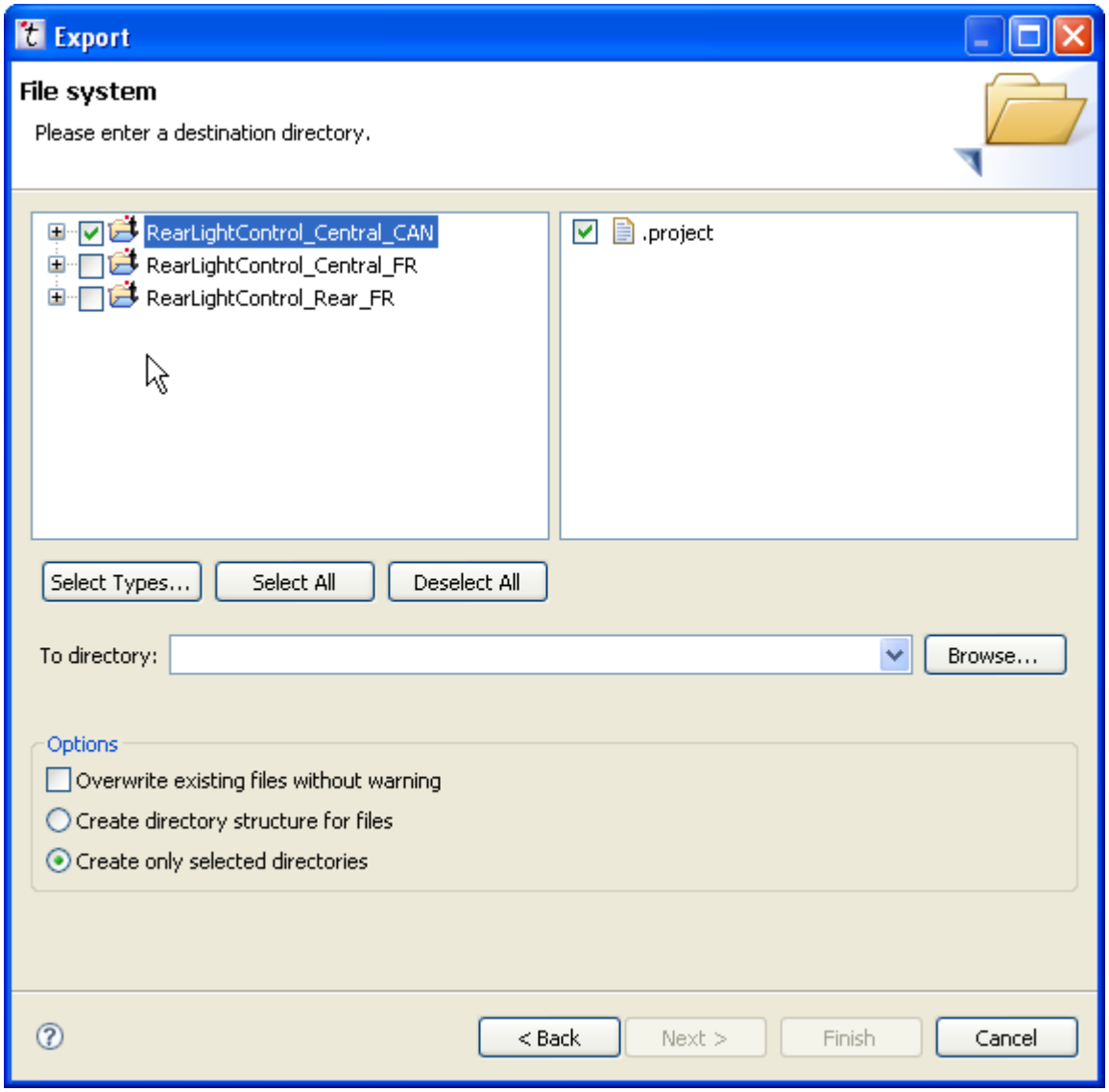
- Section 4.11.1.2.1: Exporting a project as copyexporting a project as copy,
- Section 4.11.1.2.2: Exporting a project as archiveor exporting a project as archive.

4.11.1.2.1. Exporting a project as copy

To copy the EB tresos Studio workspace data to another location, export the project as directory structure.

- In the Select window, select *File System* as export destination.
- Click the **Next** button.

  The *File system* page is shown.



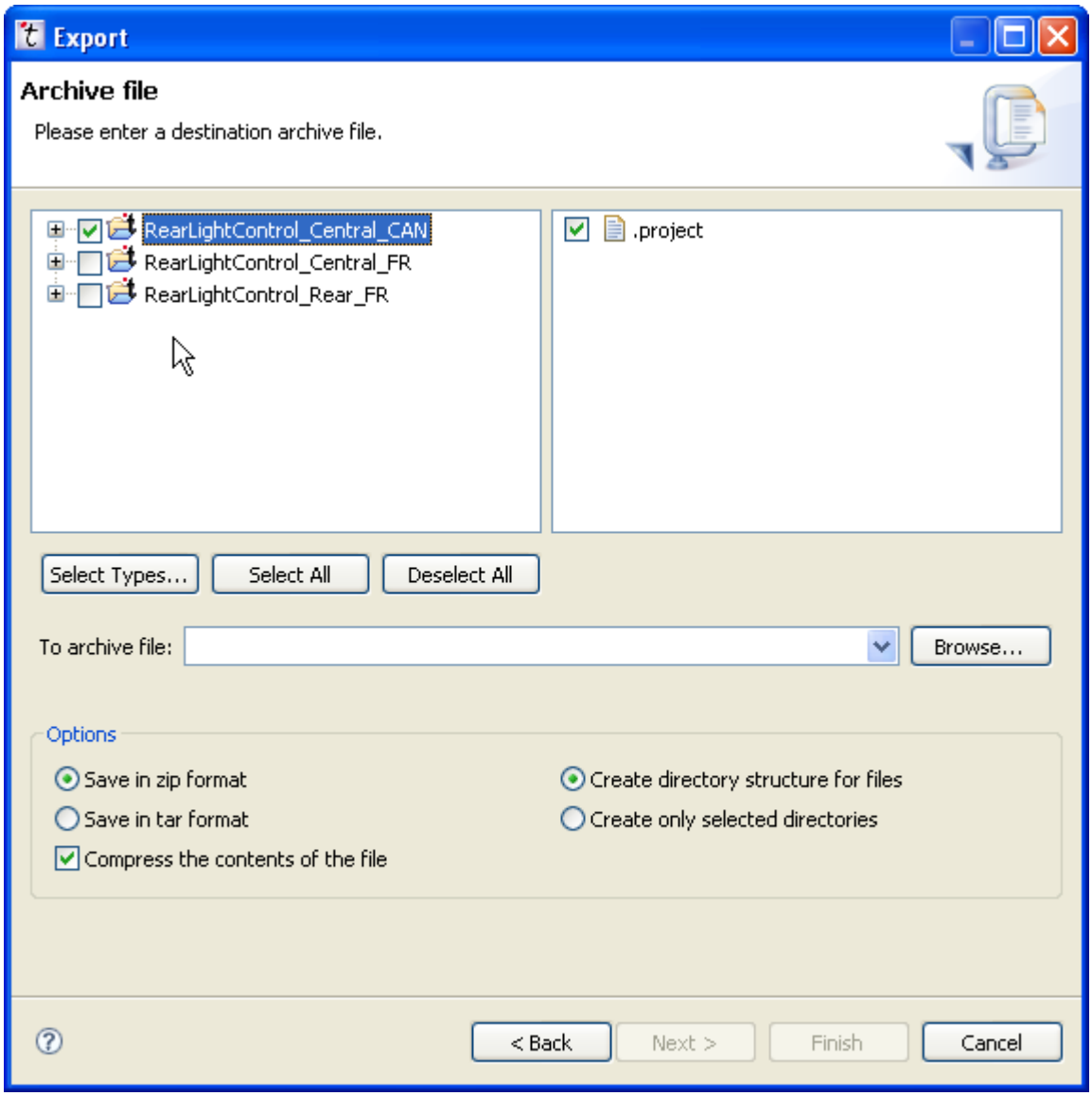
- In the list on the left of the dialog you can see all projects of your current workspace. Check all the projects you would like to export. In case you want to exclude some parts of the project, you can also expand the project in the tree and check or uncheck the contained resources. This is not recommended, as partially exported projects are usually not fully functional.
- Enter the directory into which you want to export the projects into the dropdown textbox *To directory*. You can also use the **Browse** button to choose a directory.
- In *Options*, select how you would like the project export to be performed.
- To export the projects, select **Finish**.

4.11.1.2.2. Exporting a project as archive

To store one ore more complete EB tresos Studio projects as compressed archive, export them as archive. This may be useful, e.g. if you want to exchange a project with another EB tresos Studio installation.

- In the *Select* window, select *Archive File* as export destination.
- Click the **Next** button.

  The *Archive file* page is shown.



- In the list on the left of the dialog you can see all projects of your current workspace. Check all the projects you would like to export. In case you want to exclude some parts of the project, you can also expand the project in the tree and check or uncheck the contained resources. This is not recommended, as partially exported projects are usually not fully functional.

- Enter the filename of the archive into the dropdown textbox *To archive file*. You can also use the **Browse** to choose a file.
- In *Options*, select how you would like the file system export to be performed.
- To export the projects, select **Finish**.

4.11.2. Importing and exporting configuration data

Importers and exporters are used to import and export from and to the configuration data of a project from file formats such as the Autosar ECU Configuration, OIL-format, DBC-format, LDF-format, and Fibex-format. Importing and exporting of complex project configurations includes selecting a file to store data to or load from, selecting a subset of the configuration parameters, selecting name prefixes, etc.

4.11.2.1. Step-by-step guide

Before you can export or import configuration data you need to create an export or import configuration for it. This configuration enables you to run the importer with the same settings later again. It stores all the settings you have entered for that particular import or export, and you can run that importer or exporter later again just by selecting it from a list.

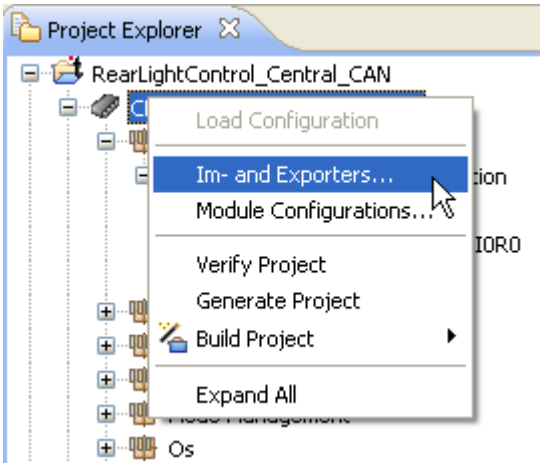
Note

Do not confuse import or export configurations with configuration projects. Importing and exporting of whole projects is covered in Section 4.11.1: Exchanging a complete EB tresos Studio project with other installations

4.11.2.1.1. Step 1: Using the Create, manage and run im- and exporters window

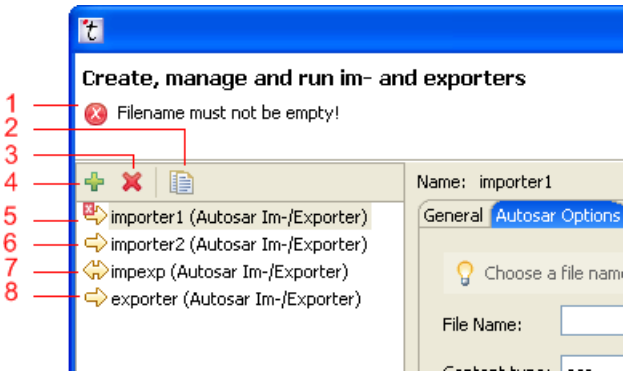
To create an import or export configuration for a configuration project:

- Select the ECU configuration node of the project you want to import to.
- Make sure your project is loaded. if not, load your project first (see Section 4.6.6.3: Loading a project .
- Right-click on the ECU configuration node.

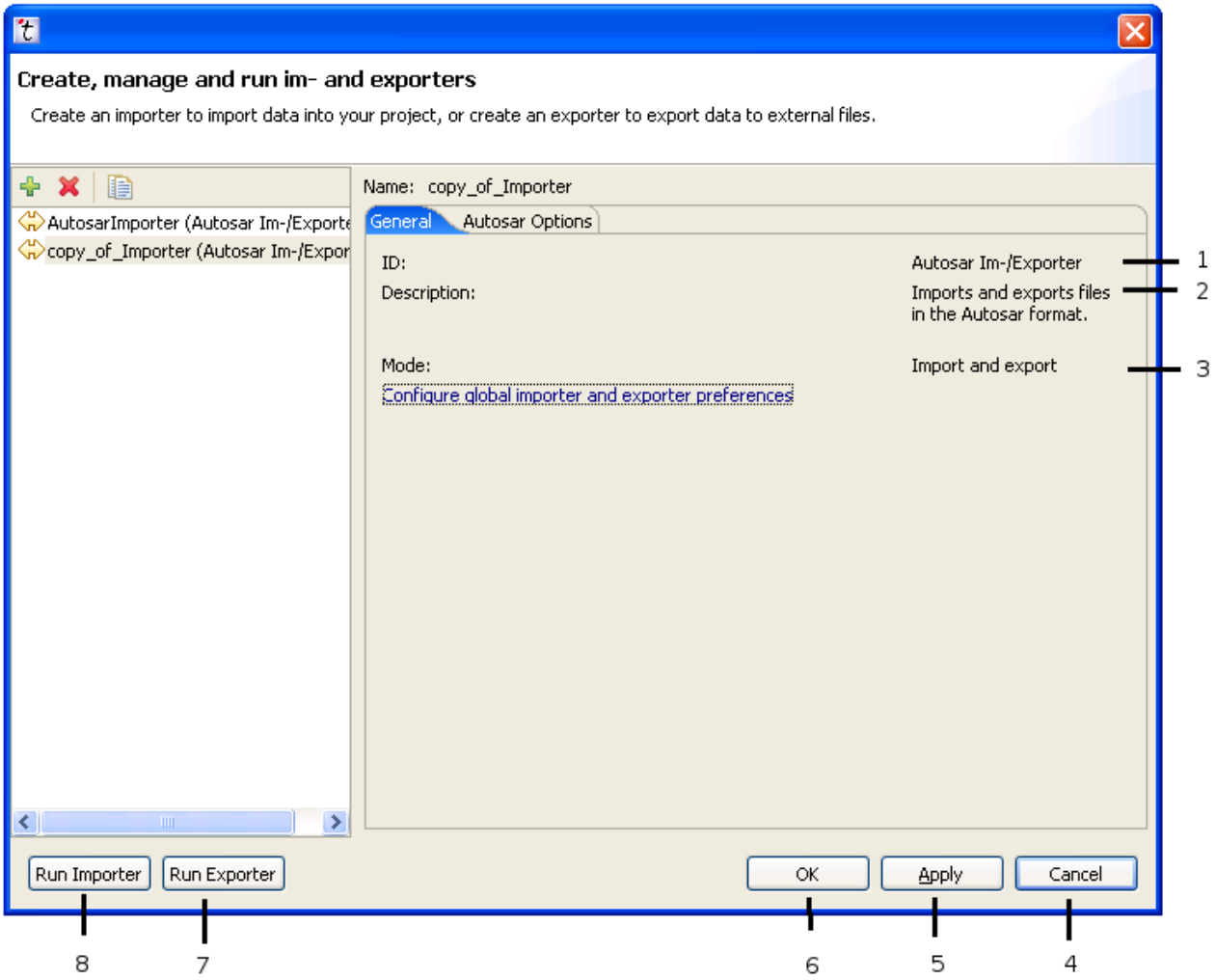


► Select *Im- and Exporters*.

The window *Create, manage and run im- and exporters* opens up.



Number	Icon	Explanation
1		If there is a problem with the importer or exporter configuration, it is shown here. If there are no problems, this section just contains a describing text.
2		Duplicate the selected import/export configuration.
3		Deletes the selected import/export configuration.
4		Adds a new import/export configuration.
5		The red cross in the top left corner indicates that this configuration contains an error.
6		This configuration will import data into EB tresos Studio.
7		This configuration will be able to do both: import and export data.
8		This configuration will export data from EB tresos Studio into other formats.



Number	Menu option/button	Explanation
1	*ID*	Type of the importer/exporter. Determines the format that is being imported or exported as well as the overall functionality of the importer/exporter.
2	*Description*	Describes the importer/exporter type that is currently selected.
3	*Mode*	The configuration mode. Modes can be: ▶ import ▶ export ▶ import and export
4	**Cancel**	Discards all changes that have been made since you opened the window or since you clicked the **Apply** button. Closes the window without asking if you want it closed.

Number	Menu option/button	Explanation
5	**Apply**	Saves changes to the settings. The window stays open.
6	**OK**	Saves changes to the settings and closes the window.
7	**Run Exporter**	Exports data into another format by using the selected export configuration.
8	**Run Importer**	Imports data into EB tresos Studio by using the selected import configuration.

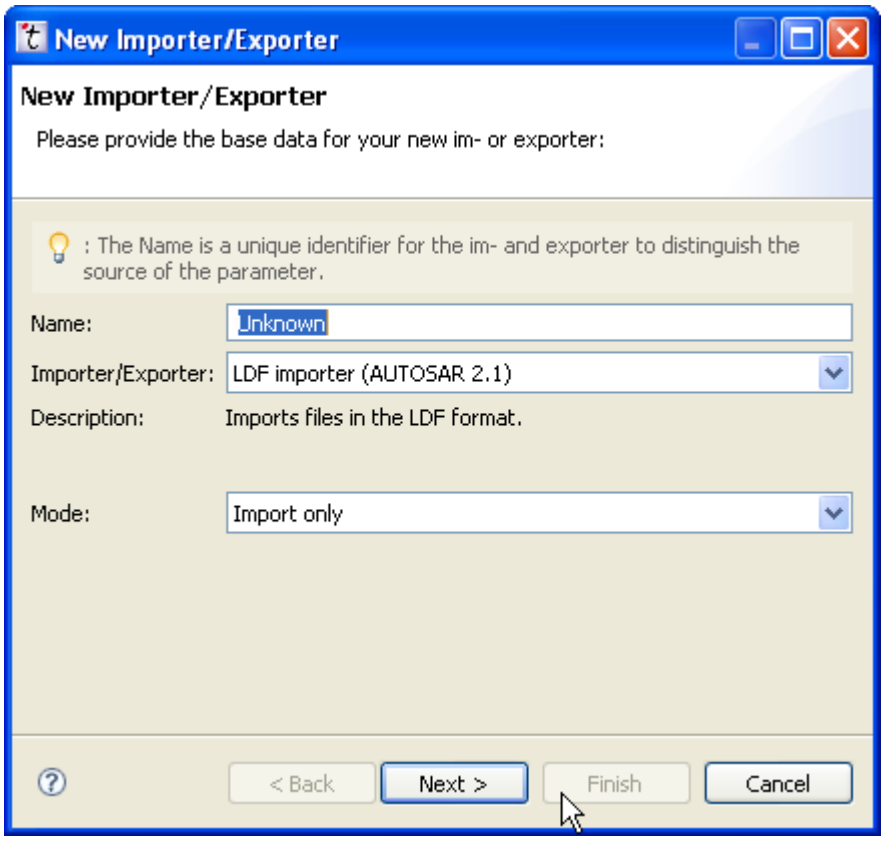
If you select one of the importer or exporter configuration on the left, the main part of the window displays the data of that configuration. The layout and the available GUI depends on the importer or exporter which you selected. All importers and exporters have an *Overview* tab which displays general information. Depending on the type, there may also other tabs be present.

4.11.2.1.2. Step 2: Creating an export or import configuration

In the window Create, manage and run im- and exporters:

- Select the **Create a new im- or exporter** button.

The *New Importer/Exporter* window opens up.

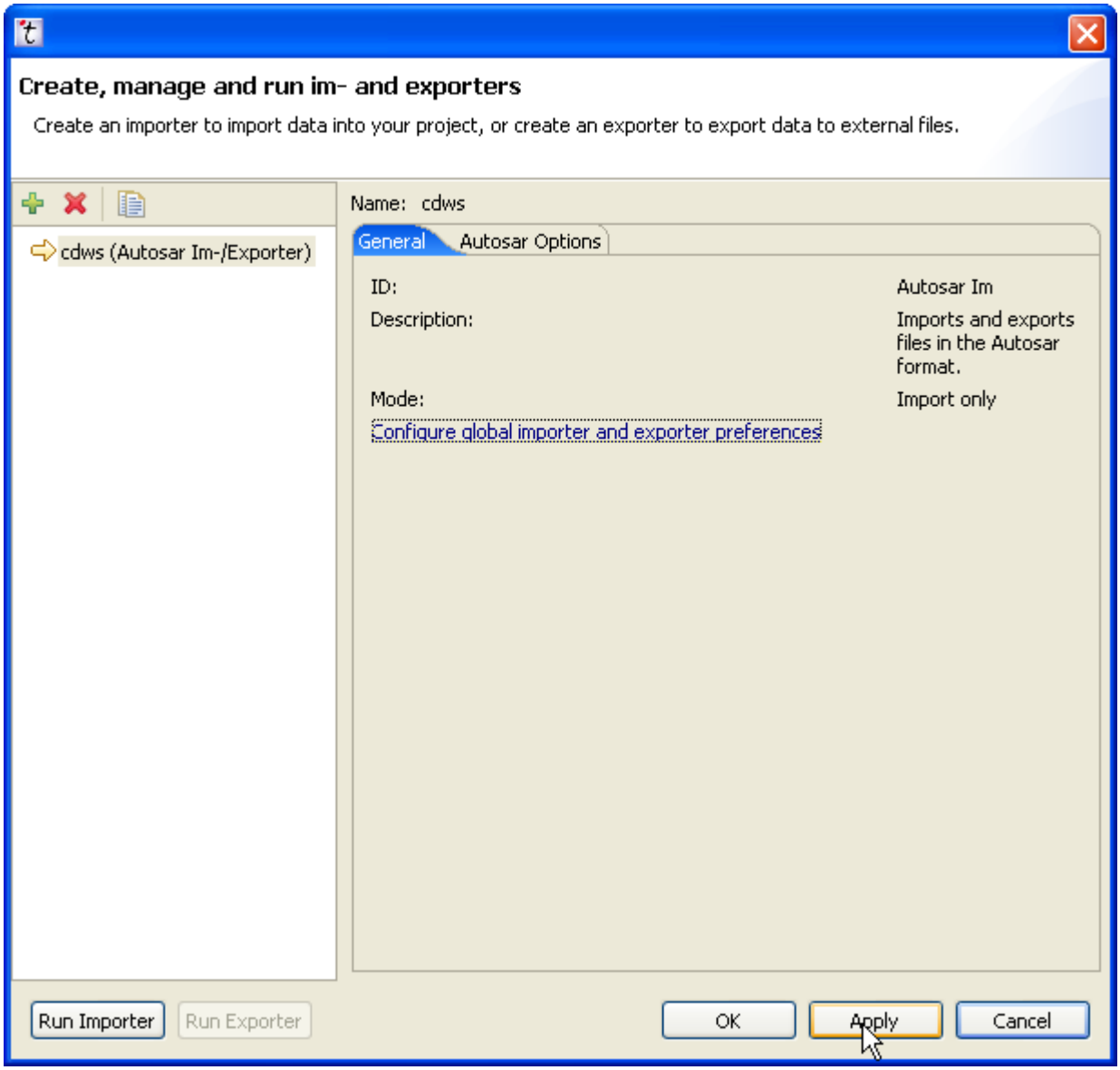


- Enter a name for your new configuration into the *Name* textbox.
- Select the type of importer or exporter from the *Importer/Exporter* dropdown listbox. The *Description* text displays a short text explaining what the selected importer or exporter does.
- Select the mode (import, export, or import and export) from the *Mode* dropdown listbox.
- Select **Next**.
- Depending on the importer or exporter, further pages may open up into which you may enter importer- or exporter-specific data.

  For further information please refer to the chapter of the respective importer or exporter.
- To switch between pages, press the **Back** button or the **Next** button.
- To finish creating the importer or exporter configuration, select the **Finish** button.
- To dismiss the entered configuration, select the **Cancel** button .

4.11.2.1.3. Step 3: Saving changes to an import or export configuration

After creating the importer or exporter configuration, you can still make changes to it in the *Create, manage and run im- and exporters* window. To save any changes you made:



- Select **Apply**.

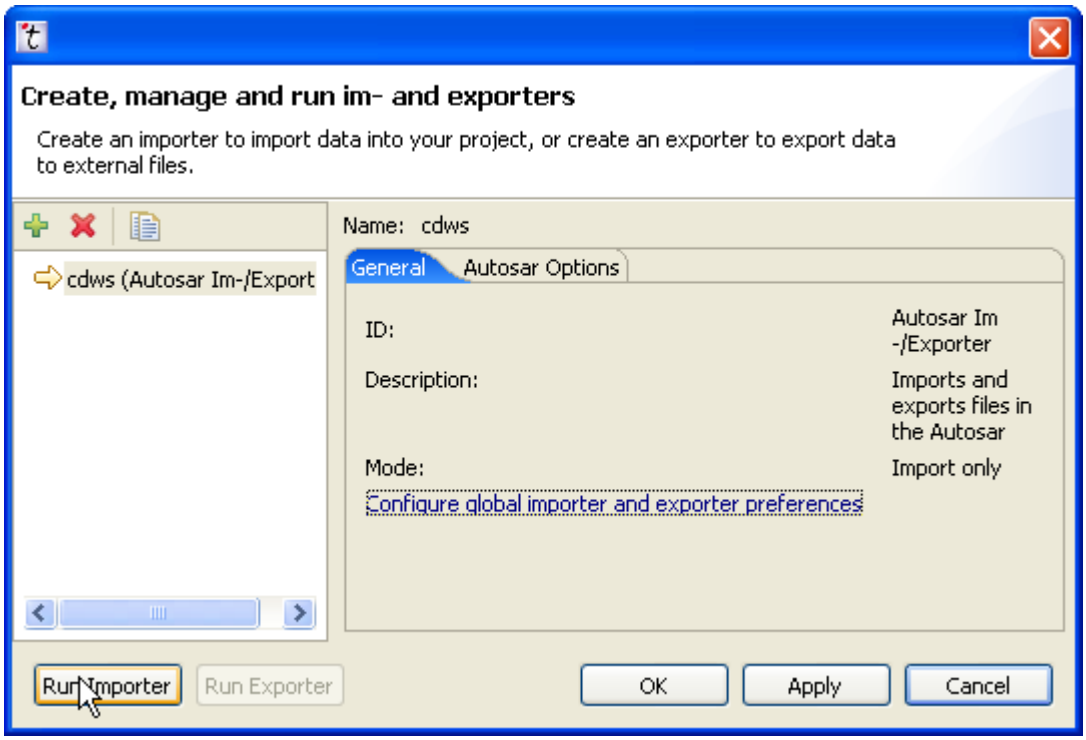
To discard changes, select **Cancel** (instead of **Apply**).

4.11.2.1.4. Step 4: Importing or exporting project configuration data

To import or export configuration data:

- Select the import or export configuration which you want to run from the list on the left side, i.e. a configuration you just created.

Depending on whether the selected configuration is set to run in import, export, or in both modes, the **Run Importers** and **Run Exporters** buttons are enabled or disabled.



- Select **Run Importer** or **Run Exporter** , depending on the task you wish to perform.

A status window opens up to inform you about the progress of your task. Then a message box provides feedback whether the importer or exporter has run successfully or not.

Possible feedback types are:

- Success
- Warning
- Error

In case of error or warning feedback, EB tresos Studio logs the feedback to the error log (see Section 4.4.5: Error Log view).

4.11.2.2. Importing and exporting configuration data from and to the Autosar format

The EB tresos Studio GUI allows you to read and write files from and to the Autosar format. For details about file content types and Autosar content types, please refer to the Background knowledge chapter Section 4.2.2.3.2: Autosar content types

4.11.2.2.1. How the import works

The configuration data imported from an Autosar file is added to the existing data of the project.

- Not yet existing elements are added.
- Already existing elements with the same Autosar SHORT-NAME path are overwritten.
- When overwriting a list of elements without SHORT-NAME (e.g. a list of integer values), each new element overwrites the existing element at the same position within the list. If the existing list contains more elements, the last elements which are not overwritten stay unchanged.
- If an importer is used a second time (or an importer with the same name), elements that have been removed from the import file are removed from the project, too.

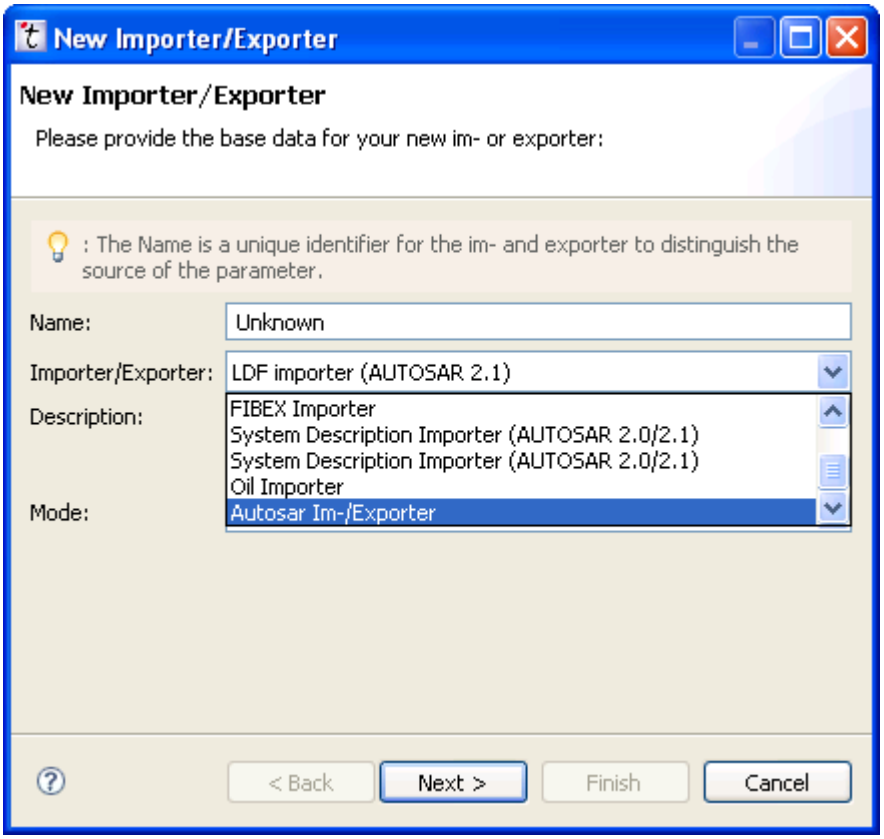
The Autosar importer is not able to do any of the following things:

- remove elements (which have not been imported before) from the project
- set an element to its default value
- enable an optional disabled element without changing its value
- disable an optional element

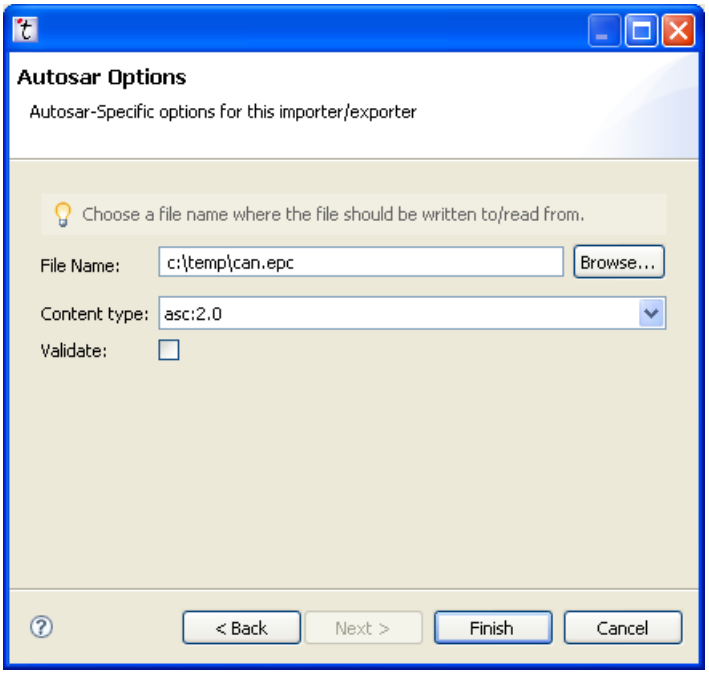
4.11.2.2.2. Importing and exporting configuration data

To import or export project configuration data in the Autosar format:

- Follow the steps in chapter Section 4.11.2: Importing and exporting configuration data to Step 2.
- In the *New Importer/Exporter* window, select the Importer/Exporter type *Autosar Im/Exporter*.



- Fill out the rest of the page according to Section 4.11.2.1: Step-by-step guide.
- Click on the **Next** button.
- The *Autosar options* page appears:



Tip

The values of this page can also be changed later in the *Create, manage and run im- and exporters* dialog.

- Enter the filename of the file on which the importer of exporter operates into the *File Name* textbox. When you click the **Browse** button a file appears that lets you choose a file.

  In case of an importer, the filename identifies the file from which the data will be imported. In case of an exporter, the filename identifies the file to which the exported data will be written to.

- Select the content type of the selected file from the *Content type* dropdown listbox. Available content types depend on the release version of your project and the importer/exporter mode.
  - *Mixed release versions*: If your project's release version is mixed, all content types are available.
  - *Import only mode*: choose a content type lower or equal to your project's release version number.
  - *Export only mode*: choose a content type equal or higher that your project's release version number.
  - *Import and export mode*: Use only the content type of your project's release version. This means if the release version of your project is e.g. 2.0 you cannot import files with content type asc:2.1 but you can export to that format.

  For more information on content types, see Section 4.2.2.3: File content types

- To enable validation, select the checkbox **Validate**:
  - When importing a file, the file will be validated against the xsd schema of the selected content type. If the file is not successfully validated, the file is not imported or exported and an error window opens up.
  - When exporting a file, the exported file will be written according to the selected content type. Additionally, the file will be validated against the xsd schema of the selected content type.

4.11.2.3. Importing Autosar system descriptions

4.11.2.3.1. Overview

EB tresos Studio provides an importer that allows to import Autosar system description (Sys-D) and software component description files (SW-CD). In this chapter you learn:

- which ECU configuration modules can be configured (from section Section 4.11.2.3.2: Background information)
- how to import system information (from section Section 4.11.2.3.3: Importing system information)

- how the ECU extract of the system description is mapped to Autosar parameters (from section Section: ECU extract to Autosar parameter mapping)

4.11.2.3.2. Background information

System information can be imported either into the system model of the project or into the ECU configuration.

The following ECU configuration modules can be configured by the importer:

- `Com` (`Communication`)
- `PduR` (`PDU Router`)
- `CanIf` (`CAN Interface`)
- `Can` (`CAN Driver`)
- `CanTp` (`CAN Transport Protocol`)
- `CanNm` (`CAN Network Management`)
- `FrIf` (`FlexRay Interface`)
- `Fr` (`FlexRay Driver`)
- `LinIf` (`LIN Interface`)
- `Lin` (`LIN Driver`)
- `EcuC` (`EcuC Module`)

Supported Autosar meta-model formats are:

- 2.1.3 (Revision 017)
- 2.1.4 (Revision 018)
- 3.0.2 (Revision 003)
- 3.1.0 (Revision 001)

When an `Rte` module is added to a EB tresos Studio project, the importer is automatically added to the project.

4.11.2.3.3. Importing system information

To import configurations from the Autosar Sys-D and SW-CD format:

- Follow the instructions of Step 1 until you arrive at the window *Create, manage and run im- and exporters*: Section 4.11.2: Importing and exporting configuration data.

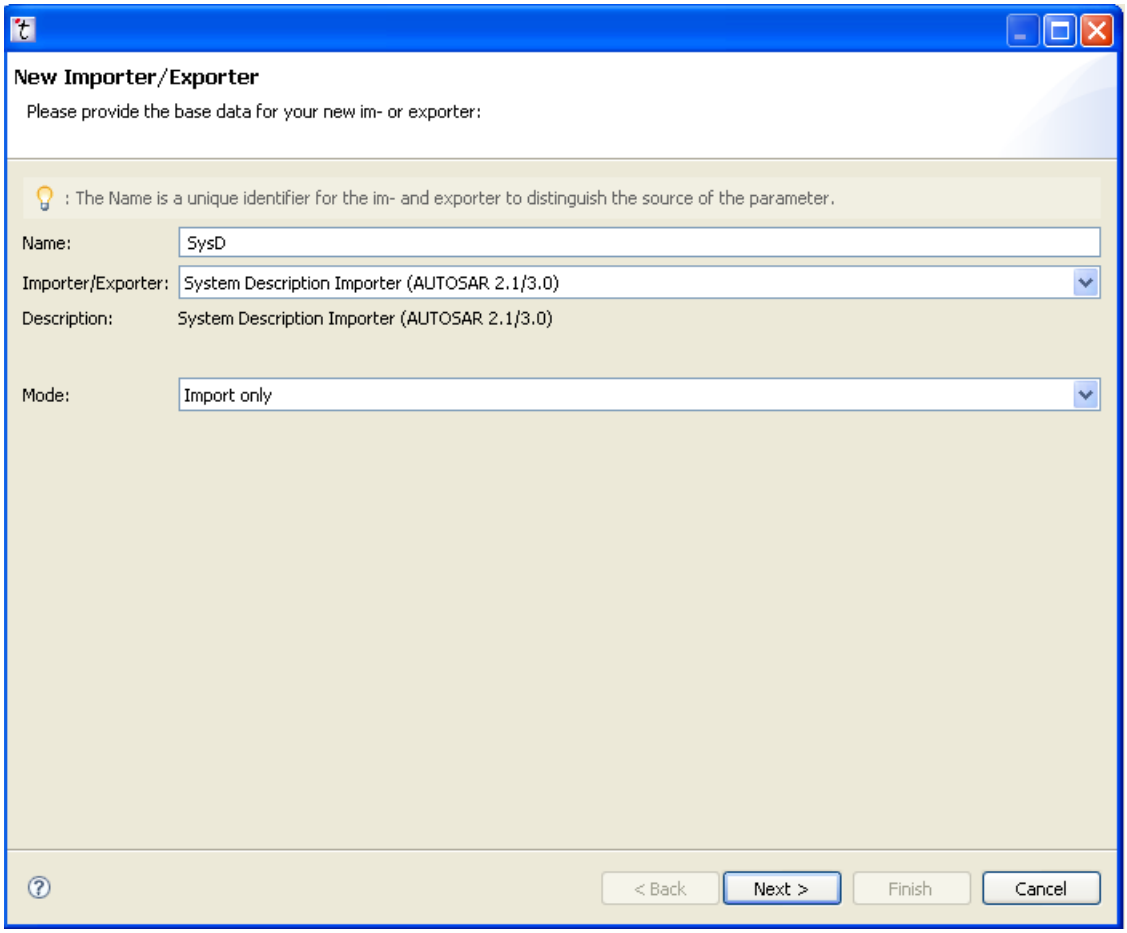
  Then return to this set of instructions.

In the window *Create, manage and run im- and exporters*:

- Select the **Create a new im- or exporter** button.

The *New Importer/Exporter* window opens up.

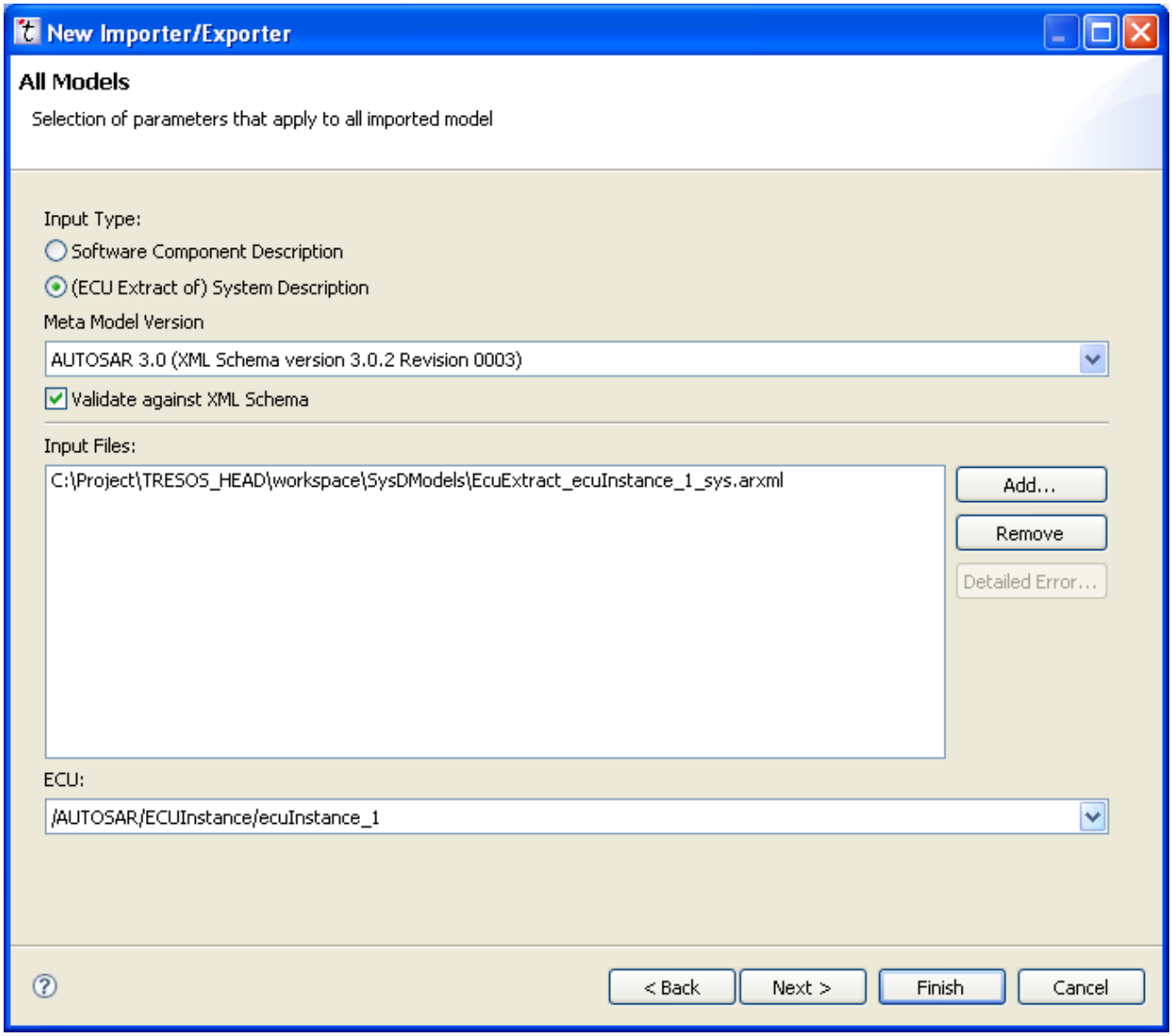
The dialog guides you through four pages as described in the following:



- Enter the name of the importer configuration into the *Name* textbox.
- Select *System Description Importer (AUTOSAR 2.1/3.0/3.1)* from the *Importer/Exporter* dropdown listbox.
- Select **Import only** from the *Mode* dropdown listbox.

The *System Description Importer* can only be used to import files. Exporting to the Autosar format is not possible.

- To proceed to the next page, press the **Next** button. On the next page you may enter generic information.



The *System Description Importer* supports two types of input files.

- To import software component descriptions only, select the *Input Type* option button *Software Component Description*.

> **`SWC-TO-ECU-MAPPING` is not evaluated**
>
> If you import the software component description via this importer, a `SWC-TO-ECU-MAPPING` is not evaluated.

- To import a system description that may include also software components, select the *Input Type* option button *(ECU Extract of) System Description*.

When a system description is imported, the importer expects that the imported files contain a `SWC-TO-ECU-MAPPING` according to the Autosar system template. The *System Description Importer* imports only software components that map to the ECU you are to select with the drop downbox **ECU** on the bottom of this window.

Note

The `SOFTWARE-COMPOSITION-TREF` of the system description is used to identify the top level composition of the system.

If the project does not include an RTE, select **(ECU Extract of) System Description** to allow importing communication stack parameters into the ECU configuration.

- To choose the Autosar version of the input files, use the dropdown listbox *Meta Model Version*.

If you change Autosar version and this creates errors, you cannot run the importer anymore

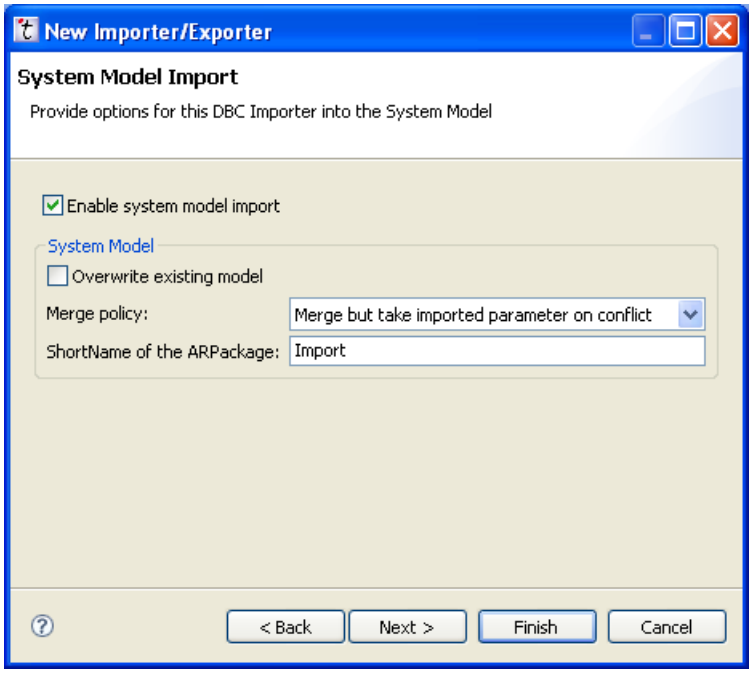
When the version is changed, selected files are parsed and some consistency checks are performed, e.g. the files are validated against the XML schema.

Errors are displayed at the top of the dialog. If an error has occurred, it is not possible to run the importer.

- To validate the imported file against the XML schema, select the **Validate against XML Schema** checkbox.
- With the **Add** and **Remove** buttons on the right of the *Input Files* textbox, you may add files to load or remove files from the list of files to be loaded.
- Use **(ECU Extract of) System Description** under *Input Type* to choose the ECU instance that you would like to imported into the ECU configuration from the *ECU* dropdown listbox .

After having selected the input files, press the **Next** button to configure the system model import.

The *System Model Import* window opens up.

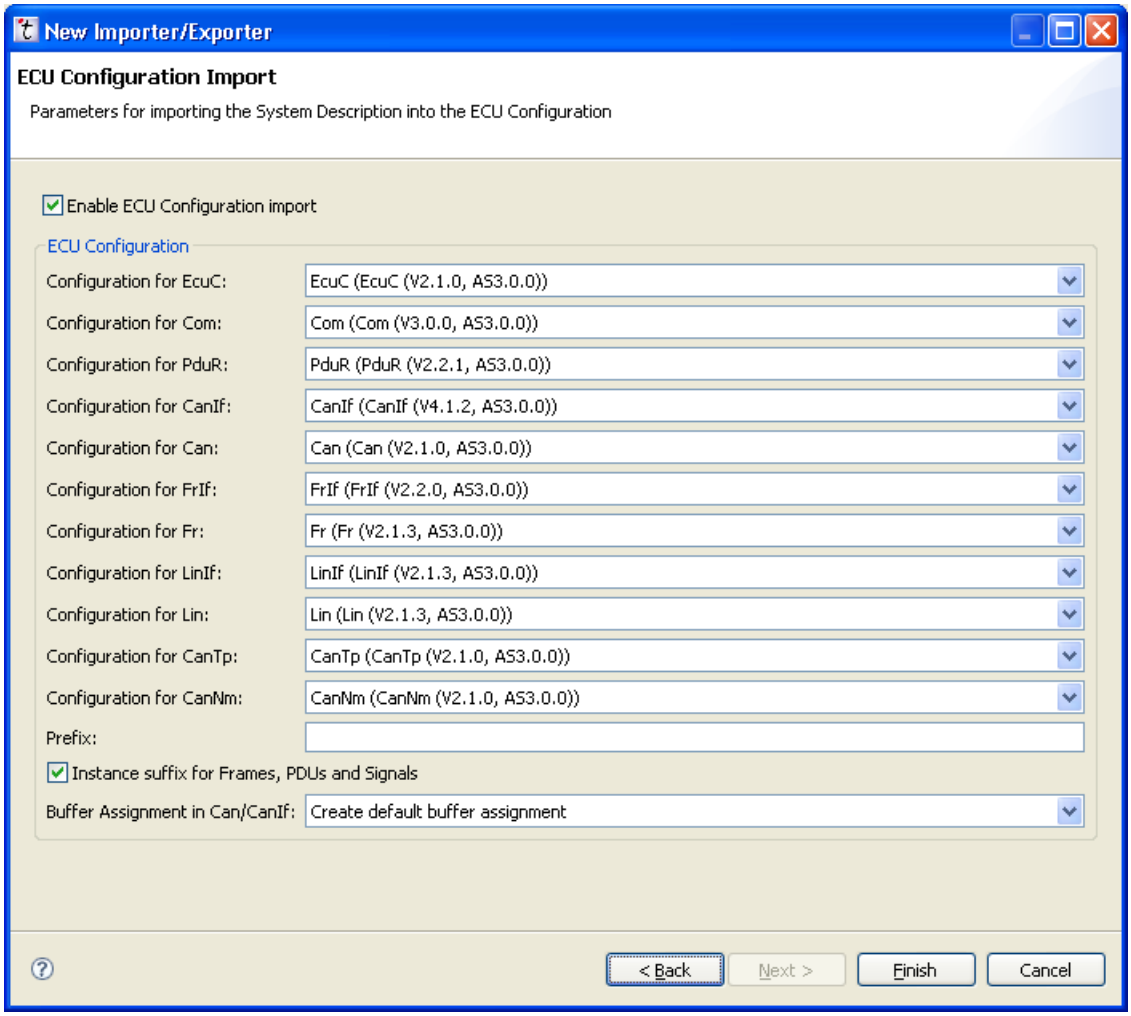


- To enable system model import into the project, check the *Enable system model import* checkbox. If the project has no system model, the complete page including the checkbox is greyed out.
- To completely replace the system model in the project, check the checkbox *Overwrite existing model*.
- To provide a hint for the merge algorithm that merges the imported configuration into the system model of the project, select an entry from the *Merge policy* dropdown listbox .
- To select the Autosar SHORT-NAME of the Autosar AR-PACKAGE into which the system model of the source file is to be imported, enter the Autosar SHORT-NAME into the **ShortName of the ARPackage** textbox.
- Select **Next** to proceed to the next configuration window.

The *ECU Configuration Import* window opens up.

Note

Since the *System Description Importer* does not create module configurations, you need to assure that a module configuration is present for every module you would like to import into (`EcuC`, `Com`, `PduR`, `CanIf`, `Can`, `FrIf`, `Fr`, `LinIf`, `Lin`, `CanTp`, `CanNm`).



- Enable or disable the import of the ECU configuration by checking or unchecking the *Enable ECU Configuration import* checkbox.
- In the *Configuration for EcuC* dropdown listbox, select the target `EcuC` configuration for the import.
- In the *Configuration for Com* dropdown listbox, select the target `Com` configuration for the import.
- In the *Configuration for PduR* dropdown listbox, select the target `PduR` configuration for the import.
- In the *Configuration for CanIf* dropdown listbox, select the target `CAN Interface` configuration for the import.
- In the *Configuration for Can* dropdown listbox, select the target `CAN Driver` configuration for the import.
- In the *Configuration for FrIf* dropdown listbox, select the target `FlexRay Interface` configuration for the import.
- In the *Configuration for Fr* dropdown listbox, select the target `FlexRay Driver` configuration for the import.
- In the *Configuration for LinIf* dropdown listbox, select the target `LIN Interface` configuration for the import.

- In the *Configuration for Lin* dropdown listbox, select the target `LIN Driver` configuration for the import.
- In the *Configuration for CanTp* dropdown listbox, select the target `CAN Transport Protocol` configuration for the import.
- In the *Configuration for CanNm* dropdown listbox, select the target `CAN Network Management` configuration for the import.
- To find out the implication of filling in the *Prefix* textbox refer to Section 4.11.3.1.2: Name prefixing.
- For details on the *Instance suffix for Frames, PDUs and Signals* checkbox see Section 4.11.3.1.3: Instance handling.
- In the *Buffer Assignment in Can/CanIf* dropdown listbox, select the buffer assignment policy for the imported Can and CanIf configuration. For detailed information, please refer to Section 4.11.2.5.3: Importing DBC configurations.

To save the importer configuration:

- Select **Finish**.

This saves your importer configuration and closes the *SysD Importer Options* window.

- To run the SysD Importer, select the importer name you just created on the left side of the *Create, manage, and run im- and exporters* window.
- Select **Run Importer**.

4.11.2.3.4. References

ECU extract to Autosar parameter mapping

Overview

The *System Description Importer* of EB tresos Studio creates configurations for the Autosar modules `Can`, `CanIf`, `CanTp`, `CanNm`, `Fr`, `FrIf`, `Lin`, `LinIf`, `PduR`, `Com`, and `EcuC`.

The parameter mappings for most modules are identical to the other communication importers. For detailed information about the parameter mappings, refer to the following sections in other chapters:

- For information on the parameter mappings of the modules `Can`, `CanIf`, and `EcuC`, please refer to Section: DBC to Autosar parameter mapping.
- For information on the parameter mappings of the modules `Fr` and `FrIf`, please refer to Section: Fibex to Autosar parameter mapping.

- For information on the parameter mappings of the modules `Lin` and `LinIf`, please refer to Section: LDF to Autosar parameter mapping.

> **Note**
>
> Please note that the *System Description Importer* currently does not support LIN frames of type `ASSIGN-FRAME-ID`, `UNASSIGN-FRAME-ID`, and `ASSIGN-NAD`.

- Information on mappings for `Com` and `PduR` can either be found in Section: DBC to Autosar parameter mapping or in Section: Fibex to Autosar parameter mapping.

For the Autosar 2.1 and Autosar 3.0 `PduR` and `Com` modules, the Autosar *System Description Importer* configures several additional parameters and configures others differently. The following sections describe the differences in detail:

- Section: Extensions to the Autosar 2.1 PduR module
- Section: Extensions to the Autosar 2.1 Com module
- Section: Autosar 3.0 PduR module
- Section: Autosar 3.0 Com module

For detailed configuration parameter descriptions, please refer to the `EB tresos AutoCore documentation`, chapter `EB tresos AutoCore module references`.

Extensions to the Autosar 2.1 PduR module

Configuration parameters	Mapping description
`PduRGlobalConfig`	`PduRConfigurationId` is set to the content of the `PDU-R-CONFIGURATION-ID` element of the corresponding `ECU-INSTANCE` element.
`PduRGlobalConfig/PduR-RoutingTable/PduRRout-ingPath`	Every PDU referenced by a `SOURCE-I-PDU` entry in the ECU instances `I-PDU-MAPPINGS` element passes the PduR gateway. For every `TARGET-I-PDU` element, an additional `PduRDestPdu` container is added to the source PDUs `PduRRoutingPath`. The `PduRDestPdu/DestPduRef` parameter refers to the destination PDUs container in the `EcuC` module. Please also note that if the ECU instance routes a source PDU from a source network to a destination PDU in a destination network, no *local* `PduRRouting-Path` for the transmission PDU on the destination network is created.

Extensions to the Autosar 2.1 Com module

The Com configuration defines several parameters that describe intervals in time. These parameters are used e.g. for time-outs or transmission patterns. All of those parameters are derived from the original interval value in the ECU extract and from the Com parameter `ComGeneral/ComConfigurationTimeBase`. The resulting intervals in the Com configurations are defined as multiples of the `ComConfigurationTimeBase`. Intervals in the ECU extract may define absolute or relative tolerances and are either `MIN` intervals that describe an interval's minimum admissible duration, or `MAX` intervals that describe an interval's maximum duration.

The formula for `MIN` intervals is `ceil((interval_duration[s]-tolerance)/ComConfigurationTimeBase[s])`. The formula for `MAX` intervals is `(interval_duration[s]+tolerance)/ComConfigurationTimeBase[s]`.

If the `ComConfigurationTimeBase` is larger than a given `MAX` interval parameter, an error is issued since Com cannot process intervals shorter than `ComConfigurationTimeBase`.

Configuration parameters	Mapping description
`ComGeneral`	`ComConfigurationTimeBase` is set to the content of the `COM-PROCESSING-PERIOD` element of the imported `ECU-INSTANCE` element. `ComConfigurationId` is set to the content of the `COM-CONFIGURATION-ID` element of the imported `ECU-INSTANCE` element.
`ComSignalGroup`	For every `SYSTEM-SIGNAL-GROUP` element which is referenced from a PDU either sent or received by the ECU instance, a `ComSignalGroup` container is created. The container name is `GR<PREFIX><name><INSTSUFFIX>`, where `<name>` is the name of the signal group as defined in the imported file. For every `ComSignalGroup` container created, a corresponding `ComNetworkSignal` container is created. `ComSignalGroupProperty` is set to `SendStaticExternal` if the signal group is sent, or to `ReceiveUnqueuedExternal` if the signal group is received. `ComSignalGroupNetworkSignalRef` is set to reference the corresponding `ComNetworkSignal` container.
`ComSignal`	For every `SYSTEM-SIGNAL` element, which is referenced from a PDU either sent or received from the ECU instance, a `ComSignal` container is created. Such containers either represent usual signals or group signals (i.e. signals which are part of signal groups). The container name is ► `SG<PREFIX><name><INSTSUFFIX>`, where `<name>` is the name of the signal as defined in the imported file, ► or `GS<PREFIX><name><INSTSUFFIX>`, where `<name>` is the name of the group signal as defined in the imported file .

Configuration parameters	Mapping description
	For every `ComSignal` container created , a corresponding `ComNetworkSignal` container is created. The following paragraphs list the parameters, which are additionally added by the *System Description Importer*. ► `ComSignalInitValue` is set to the content of the referenced `INIT-VALUE` if the value is either defined using an `INTEGER-LITERAL`, a `BOOLEAN-LITERAL`, or an `ARRAY-SPECIFICATION`. ► `ComSignalGroupRef` is set to reference the corresponding `ComSignalGroup`, if the `ComSignal` container represents a group signal. ► `ComSignalType` is set to ► `UInt8|16|32` or to `SInt8|16|32`, according to the lower and upper bounds if the data type assigned to the corresponding system signal is of type `IntegerType`. ► `Boolean` if the data type assigned to the corresponding system signal is of type `BooleanType`. ► `UInt8_n` in all other cases.
`ComSignal/ComSignalRouteDest`	If any `SIGNAL-MAPPING` entry assigned to the ECU instance references the signal or signal group via the `SOURCE-SIGNAL-REF` reference, a list of `ComSignalRouteDest` containers is created. For every signal or signal group,which is referenced in the `TARGET-SIGNAL-REF` entries of the `SIGNAL-MAPPING`, one container is added to the `ComSignalRouteDest` container list. The container parameter `ComSignalRouteDestRef` is set to link to the signal or signal group container that has been added for the element referenced by `TARGET-SIGNAL-REF`.
`ComNetworkSignal`	For every signal or signal group created in the Com configuration, an additional `ComNetworkSignal` container is created with the name `NW<PREFIX><name><INSTSUFFIX>`, where `<name>` is the name of the corresponding signal or signal group. ► `ComNetworkSignalBitPosition` is set to the content of the `START-POSITION` element. ► `ComNetworkSignalDirection` is set to `Sent`, if the corresponding signal/signal group is sent, else to `Received`. ► `ComNetworkSignalEndianess` is set to `Opaque` if the network signal corresponds to a signal group or to a signal with `ComSignalType UInt8_n`. In all other cases, the parameter is set to `BigEndian` or to `LittleEndian`, depending on the endianess of the imported signal.

Configuration parameters	Mapping description
	▶ ComNetworkSignalSize is set to the number of bits of the corresponding signal or, if the network signal corresponds to a signal group, to the number of bits necessary to completely cover the data range of all contained group signals. ▶ ComNetworkSignalTimeoutFact is calculated from the TIMEOUT value of the corresponding signal port according to the interval calculation formula defined above. ▶ ComNetworkSignalIPduRef is set to reference the enclosing I-PDU. ▶ ComUpdate/ComUpdateBitPosition is set to the bit offset of the update bit of the signal/signal group within the enclosing PDU.
ComIPdu	For every SIGNAL-I-PDU element, which is either sent or received by the ECU instance, a ComIPdu container is created. The container name is PD<PREFIX><name><INSTSUFFIX>, where <name> is the name of the PDU as defined in the imported file. The following paragraphs list the parameters which are additionally added by the *System Description Importer*. ▶ ComIPduUnusedAreasDefault is set to the content of the PDUs UNUSED-BIT-PATTERN element. ▶ ComIPduMinimumDelayFactor is calculated as MIN parameter from the content of MINIMUM-DELAY according to the interval calculation formula above. This parameter is only configured for sent ComIPdus.
ComIPdu/ComTransmissionMode-True	The following parameters are only configured for sent I-PDUs. ComTxModeMode is set to one of the following values for outgoing ComIPdus : ▶ Direct, if only one EVENT-CONTROLLED-TIMING is defined in the PDUs I-PDU-TIMING-SPECIFICATION. ▶ Periodic, if only one CYCLIC-TIMING is defined in the PDUs I-PDU-TIMING-SPECIFICATION. ▶ Mixed, if both EVENT-CONTROLLED-TIMING and CYCLIC-TIMING are defined in the PDUs I-PDU-TIMING-SPECIFICATION. If ComTxModeMode is set to Direct, the MIN parameter ComTxModeRepetitionPeriodFactor is calculated from the DEBOUNCE-TIME value according to the interval calculation formula above. ComTxModeNumberOfRepetitions is set to NUMBER-OF-REPEATS. If ComTxModeMode is set to Periodic, the MAX parameter ComTxModeTimePeriodFactor is calculated from the REPEATING-TIME value according to the interval calculation formula

Configuration parameters	Mapping description
	above. The MIN parameter ComTxModeTimeOffsetFactor is derived from the STARTING-TIME value. ComTxModeNumberOfRepetitions is set to NUMBER-OF-REPEATS. If ComTxModeMode is set to Mixed, all parameters for Direct and Cyclic mode are set.
ComConfig/ComIPduGroup	If the imported file contains any I-PDU-GROUPs, which are referenced by the imported ECU-INSTANCE, these I-PDU-GROUPs are imported as ComConfig/ComIPduGroup containers. If the imported file does not contain any of these I-PDU-GROUPs, the default ComConfig/ComIPduGroup containers described in e.g. Section 4.11.2.5.3: Importing DBC configurations are created instead.

Autosar 3.0 PduR module

The extensions for the Autosar 3.0 PduR module are the same as for the Autosar 2.1 PduR module. Refer to Section: Extensions to the Autosar 2.1 PduR module for details.

Autosar 3.0 Com module

The Com configuration defines several parameters describing intervals in time, which are used e.g. for timeouts or transmission patterns. Please refer to Section: Extensions to the Autosar 2.1 Com module for details.

Configuration parameters	Mapping description
ComGeneral	ComConfigurationTimeBase is set to the content of the COM-PROCESSING-PERIOD element of the imported ECU-INSTANCE element.
ComConfig	ComConfigurationId is set to the content of the COM-CONFIGURATION-ID element of the imported ECU-INSTANCE element.
ComConfig/ComSignalGroup	For every SYSTEM-SIGNAL-GROUP element, which is referenced from a PDU either sent or received by the ECU instance, a ComSignalGroup container is created. The container name is GR<PREFIX><name><INSTSUFFIX>, where <name> is the name of the signal group as defined in the imported file. ComBitPosition is set to the content of the START-POSITION element. ComBitSize is set to the number of bits which are necessary to cover the data range of all group signals contained therein.
ComConfig/ComSignalGroup/ComGroupSignal	For every signal contained in sent or received SYSTEM-SIGNAL-GROUPs, one ComGroupSignal subcontainer is created within the enclosing ComGroupSignal container.

Configuration parameters	Mapping description
	The container name is GS<PREFIX><name><INSTSUFFIX>, where <name> is the name of the group signal as defined in the imported file. The exported parameters are the same as the parameters for a ComSignal, except for the ComUpdateBitPosition and the ComTimeoutFactor parameters, which are not defined within a ComGroupSignal container.
ComConfig/ComSignal ComConfig/ComSignalGroup	ComUpdateBitPosition is set to the content of the UPDATE-INDICATION-BIT-POSITION element of the corresponding I-SIGNAL-TO-I-PDU-MAPPING element. ComTimeoutFactor is calculated from the corresponding signal ports TIMEOUT value according to the interval calculation formula outlined in Section: Extensions to the Autosar 2.1 Com module.
ComConfig/ComSignal ComConfig/ComSignalGroup ComConfig/ComSignalGroup/ComGroupSignal	SystemTemplateSystemSignalRef is set to the corresponding ISignalToIPduMapping container of the imported system model.
ComConfig/ComSignal ComConfig/ComSignalGroup/ComGroupSignal	ComSignalInitValue is set to the content of the INIT-VALUE referenced, if the value is either defined using an INTEGER-LITERAL, a BOOLEAN-LITERAL, or an ARRAY-SPECIFICATION. ComSignalType is set to ▶ UINT8|16|32 or to SINT8|16|32, according to the lower and upper bounds if the data type assigned to the corresponding system signal is of type IntegerType. ▶ BOOLEAN if the data type assigned to the corresponding system signal is of type BooleanType. ▶ UINT8_N in all other cases. ComSignalEndianess is set to OPAQUE if the the ComSignalType has been set to UINT8_N. In all other cases, the parameter is set to BIG_ENDIAN or to LITTLE_ENDIAN, depending on the endianess of the imported signal or group signal.
ComConfig/ComIPdu/ComTxIPdu	The following parameters are only configured for sent I-PDUs:

Configuration parameters	Mapping description
	The MIN parameter ComTxIPduMinimumDelayTimeFactor is calculated from the content of MINIMUM-DELAY according to the interval calculation formula outlined in Section: Extensions to the Autosar 2.1 Com module. ComTxIPduUnusedAreasDefault is set to the content of the PDUs UNUSED-BIT-PATTERN element.
ComConfig/ComIPdu/ComTxIPdu/ComTxModeTrue	ComTxModeMode is set to one of the following values for outgoing ComIPdus: ▶ DIRECT, if only one EVENT-CONTROLLED-TIMING is defined in the PDUs I-PDU-TIMING-SPECIFICATION. ▶ PERIODIC, if only one CYCLIC-TIMING is defined in the PDUs I-PDU-TIMING-SPECIFICATION. ▶ MIXED, if both EVENT-CONTROLLED-TIMING and CYCLIC-TIMING are defined in the PDUs I-PDU-TIMING-SPECIFICATION. If ComTxModeMode is set to DIRECT, the MIN parameter ComTxModeRepetitionPeriodFactor is calculated from the DEBOUNCE-TIME value according to the interval calculation formula outlined in Section: Extensions to the Autosar 2.1 Com module. ComTxModeNumberOfRepetitions is set to NUMBER-OF-REPEATS. If ComTxModeMode is set to PERIODIC, the MAX parameter ComTxModeTimePeriodFactor is calculated from the REPEATING-TIME value according to the interval calculation formula outlined in Section: Extensions to the Autosar 2.1 Com module. The MIN parameter ComTxModeTimeOffsetFactor is derived from the STARTING-TIME value. ComTxModeNumberOfRepetitions is set to NUMBER-OF-REPEATS. If ComTxModeMode is set to MIXED, all parameters for DIRECT and CYCLIC mode are set.
ComConfig/ComIPduGroup	If the Sys-D file contains any I-PDU-GROUPs, which are referenced by the imported ECU-INSTANCE, these I-PDU-GROUPs are imported as ComConfig/ComIPduGroup containers. If the Sys-D file does not contain any of these I-PDU-GROUPs, the default ComConfig/ComIPduGroup containers described in e.g. Section 4.11.2.5.3: Importing DBC configurations are created instead.
ComConfig/ComGatewayMapping	For every SIGNAL-MAPPING entry assigned to the ECU instance, one ComGatewayMapping container is created. The container name is GM<PREFIX><name><INSTSUFFIX>, where <name> is the name of the signal referenced in the SOURCE-SIGNAL-REF element. ComGwSource is set to ComGwSignal. ComGwSource/ComGwSignalRef references the source signals ComSignal container. For every TARGET-SIGNAL-REF,

Configuration parameters	Mapping description
	one ComGwDestination container is added, the ComGwDestination/ComGwSignalRef reference is set to the destination signals ComSignal container.

Autosar 3.0 CanTp module

Configuration parameters	Mapping description
CanTpRxNSdu	For every CAN-TP-CONNECTION-CHANNEL element, which references a PDU received by the ECU instance via TP-SDU-REF, a CanTpRxNSdu container is created. The container name is <PREFIX><name>, where <name> is the name of PDU referenced by the TP-SDU-REF as defined in the imported file. CanTpAddressingFormat is set depending on the ADDRESSING-FORMAT: ▶ CANTP_EXTENDED for EXTENDED. ▶ CANTP_STANDARD for STANDARD. CanTpBs is set to BLOCK-SIZE. CanTpSTMin is calculated from MINIMUM-SEPARATION-TIME for the following intervals: ▶ [0us] is mapped onto 0x00. ▶ (0us..900us] is mapped onto 0xF1..0xF9, where (0us..100us] maps to 0xF1, (100us..200us] maps to 0xF2, ... (800us..900us] maps to 0xF9. ▶ (0.9ms..127ms] is mapped onto the next higher integral value. CanTpRxDl is set to the bytelength of the PDU referenced by the TP-SDU-REF. CanTpRxNSduRef is set to the path of the corresponding PDU in the EcuC (TP-SDU-REF) CanTpRxPaddingActivation is set depending on the content of PADDING-ACTIVATION: ▶ CANTP_ON for TRUE. ▶ CANTP_OFF for FALSE. CanTpRxTaType is set depending on the MULTICAST-ADDRESSING: ▶ CANTP_FUNCTIONAL for TRUE.

Configuration parameters	Mapping description
	▶ CANTP_PHYSICAL for FALSE.
CanTpRxNSdu/CanTpNSa	CanTpNSa is set to the TP-ADDRESS of the COMMUNICATION-CONNECTOR referenced by SOURCE-REF if the ADDRESSING-FORMAT is EXTENDED.
CanTpRxNSdu/CanTpNTa	CanTpNTa is set to the TP-ADDRESS of the COMMUNICATION-CONNECTOR referenced by TARGET-REF if the ADDRESSING-FORMAT is EXTENDED.
CanTpRxNSdu/CanTpRxNPdu	CanTpRxNPduRef is set to the path of the corresponding PDU in the EcuC (DATA-PDU-REF).
CanTpRxNSdu/CanTpTxFcNPdu	CanTpTxFcNPduRef is set to the path of the corresponding PDU in the EcuC (FLOW-CONTROL-PDU-REF).
CanTpTxNSdu	For every CAN-TP-CONNECTION-CHANNEL element, which references a PDU sent by the ECU instance via TP-SDU-REF, a CanTpTxNSdu container is created. The container name is <PREFIX><name>, where <name> is the name of PDU referenced by the TP-SDU-REF as defined in the imported file. CanTpAddressingMode is set depending on the ADDRESSING-FORMAT: ▶ CANTP_EXTENDED for EXTENDED. ▶ CANTP_STANDARD for STANDARD. CanTpTxDl is set to the bytelength of the PDU referenced by the TP-SDU-REF. CanTpTxNSduRef is set to the path of the corresponding PDU in the EcuC (TP-SDU-REF). CanTpTxPaddingActivation is set depending on the PADDING-ACTIVATION: ▶ CANTP_ON for TRUE. ▶ CANTP_OFF for FALSE. CanTpTxTaType is set depending on the MULTICAST-ADDRESSING: ▶ CANTP_FUNCTIONAL for TRUE. ▶ CANTP_PHYSICAL for FALSE.
CanTpTxNSdu/CanTpNSa	CanTpNSa is set to the TP-ADDRESS of the COMMUNICATION-CONNECTOR referenced by SOURCE-REF if the ADDRESSING-FORMAT is EXTENDED.
CanTpTxNSdu/CanTpNTa	CanTpNTa is set to the TP-ADDRESS of the COMMUNICATION-CONNECTOR referenced by TARGET-REF if the ADDRESSING-FORMAT is EXTENDED.

Configuration parameters	Mapping description
CanTpTxNSdu/CanTpTxNPdu	CanTpTxNPduRef is set to the path of the corresponding PDU in the EcuC (DATA-PDU-REF).
CanTpTxNSdu/CanTpRxFcNPdu	CanTpRxFcNPduRef is set to the path of the corresponding PDU in the EcuC (FLOW-CONTROL-PDU-REF).

Autosar 3.0 CanNm module

Configuration parameters	Mapping description
CanNmGlobalConfig	The following parameters are set for the first CAN-CLUSTER, which sends/receives at least one Rx and/or Tx NM-PDU: CanNmBusLoadReductionEnabled is set to NM-BUS-LOAD-REDUCTION-ENABLED. CanNmNodeDetectionEnabled is set to NM-NODE-DETECTION-ENABLED. CanNmNodeIdEnabled is set to NM-NODE-ID-ENABLED. CanNmRepeatMsgIndEnabled is set to NM-REPEAT-MESSAGE-SUPPORT.
CanNmGlobalConfig/CanNmChannelConfig	Parameters are set for each PHYSICAL-CHANNEL of a CAN-CLUSTER, which sends/receives at least one Rx and/or Tx NM-PDU. The following parameter values are taken from the Rx NM-PDU, or, if no Rx NM-PDU is defined, from the Tx NM-PDU. CanNmPduLength is set to the PDU's bytelength. CanNmPduCbvPosition is set depending on the NM-CBV-POSITION. ▶ CANNM_PDU_BYTE_0 for values ranging from 0 to 7. ▶ CANNM_PDU_BYTE_1 for values ranging from 8 to 15. ▶ CANNM_PDU_OFF if the value is not defined or for any different value. CanNmPduNidPosition is only set if the cluster's parameter NM-NODE-ID-ENABLED is set to TRUE. The parameter is set depending on NM-NID-POSITION: ▶ CANNM_PDU_BYTE_0 for values ranging from 0 to 7. ▶ CANNM_PDU_BYTE_1 for values ranging from 8 to 15. ▶ CANNM_PDU_OFF if the value is not defined or for any different value. CanNmUserDataLentgh (sic) is set to NM-USER-DATA-LENGTH.

Configuration parameters	Mapping description
	CanNmNodeId is set to NM-ADDRESS of the first COMMUNICATION-CONNECTOR connecting the ECU to this PHYSICAL-CHANNEL. The following parameter values are taken from the CAN-CLUSTER connected to the PHYSICAL-CHANNEL. CanNmBusLoadReductionActive is set to NM-BUS-LOAD-REDUCTION-ACTIVE. CanNmRepeatMessageTime is set to NM-REPEAT-MESSAGE-STATE-TIME. CanNmTimeoutTime is set to NM-TIMEOUT-TIME. CanNmWaitBusSleepTime is set to NM-WAIT-BUS-SLEEP-TIME.
CanNmGlobalConfig/CanNmChannelConfig/CanNmTxPdu	The reference to the EcuC PDU container is set here (Tx NM-PDU): CanNmTxPduRef.
CanNmGlobalConfig/CanNmChannelConfig/CanNmRxPdu	The reference to the EcuC PDU container is set here (Rx NM-PDU): CanNmRxPduRef.

4.11.2.4. Importing from the OIL format

4.11.2.4.1. Overview

EB tresos Studio provides an integrated OIL Importer to import Os configurations from OIL files. If an OIL file is available for import, the OIL Importer simplifies and speeds up the configuration of the Os module. It is also possible to import OIL files via the command line. This chapter describes the tool's support for the OIL format from the GUI and the command line.

4.11.2.4.2. OIL Importer and file format background knowledge

The OIL Importer can read an OIL file and is able to configure the Os module.

The OIL Importer creates default configurations for hardware-related options. It creates one Os configuration.

During the import, the OIL Importer creates several containers for frames and signals. The names chosen for these containers are based on the names defined in the OIL file.

The mapping of the OIL format to the Autosar parameters is described in the chapter The Oil attribute naming translation of the Os module documentation.

4.11.2.4.3. Importing OIL configurations

Since the OIL Importer does not create module configurations, you need to assure that an Os module configuration is present.

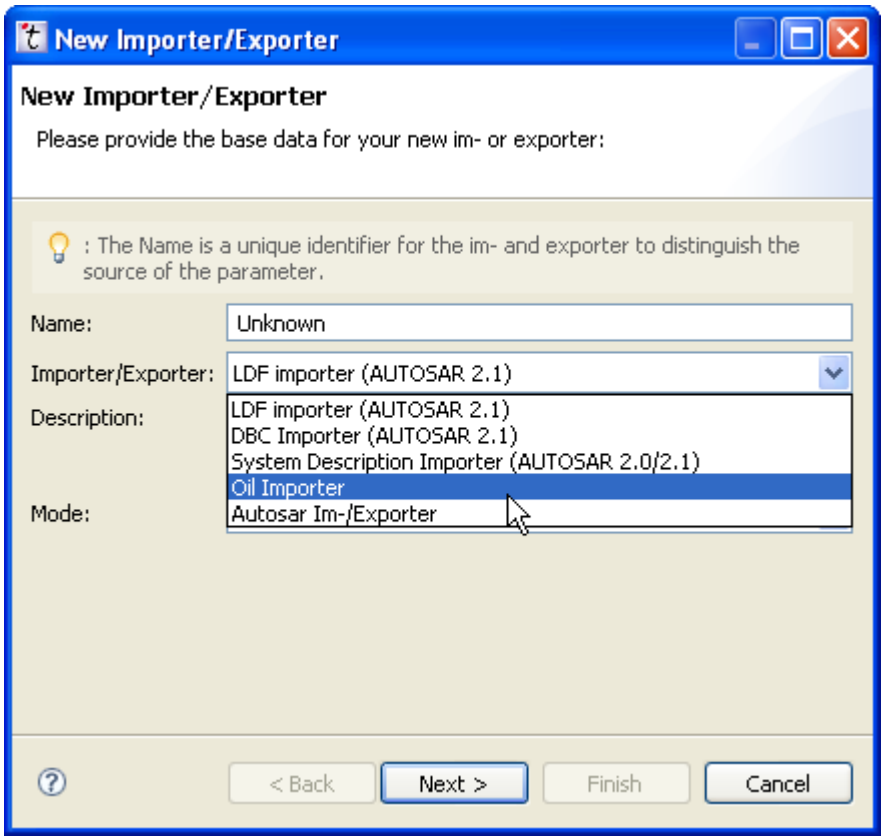
To import a configuration from the OIL format:

- Refer to Section 4.11.2: Importing and exporting configuration data.
- Follow the instructions of Step 1 until you arrive at the window *Create, manage and run im- and exporters*.

In the window Create, manage and run im- and exporters:

- Select the **Create a new im- or exporter** button.

The *New Importer/Exporter* window opens up.



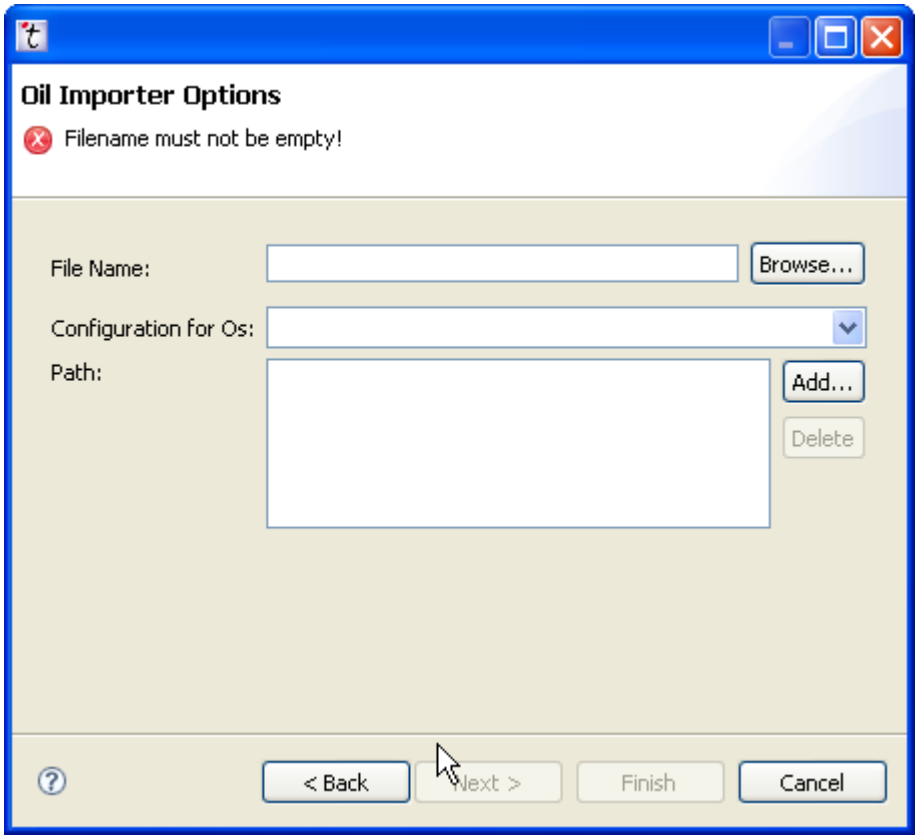
To enter the basic information for the new importer or exporter configuration:

- Select the *Name* textbox and enter a name for your new configuration.
- Select *OIL Importer* from the **Importer/Exporter** dropdown listbox.
- Select the *Mode* line dropdown listbox.
- Select **Import only** from the dropdown listbox.

The OIL Importer can only be used to import files. Exporting to OIL format is not possible.

- Click the **Next** button.

The *OIL Importer Options* page is shown.



- In the *File Name* textbox browse for the OIL file you want to import into EB tresos Studio.
- In the *Configuration for Os* menu line, select the target Os configuration for the import.
- In the *List of Path names* menu line, select a list of path names as include paths to additional OIL files.

To save the importer configuration:

- Select **Finish**.

  This saves your importer configuration and closes the *OIL Importer Options* window.

To run the OIL Importer, select the file name you just created on the left side of the *Create, manage, and run im- and exporters* window.

- Select **Run Importer**.

  The OIL configuration is being imported.

4.11.2.4.4. Importing OIL files via the command line

The OIL Importer of EB tresos Studio can also be used on the command line, e.g. to import OIL files in batch mode. This import function is encapsuled into the **legacy convert** command.

You may need to edit the configuration that has been created during the import before you use it to configure an ECU.

OIL Importer parameters for the command line

The commandline OIL Importer requires passing some system properties to the command line via the `-D` option. These are:

- `target` (mandatory): Specifies the target for which to import.
- `oil.include` (optional): List of pathnames, where possible oil include files are located.
- `oil.package` (mandatory) ARPackage name of the selected Os configuration.
- `oil.config` (mandatory) ConfigName of the Os configuration selected .

OIL import example

The following example shows a typical use of the OIL Importer (all on one line):

```
tresos_cmd.bat  -Dtarget=V850
  -Doil.include=C:\;D:\parser
  -Doil.package=Os
  -Doil.config=Os
  -Dmsg1648=false legacy convert
  network.oil output.epc
```

This line will import all frames and signals from the file `network.oil` into the Os module for the V850 architecture.

The `-Dmsg1648=false` parameter disables a warning message that would otherwise appear. The warning - if not disabled - informs users that all nodes that are required are generated automatically. Since this applies only to nodes that are not extracted from the OIL file, you may safely ignore this warning.

4.11.2.5. Importing CAN configurations from the DBC format

4.11.2.5.1. Overview

EB tresos Studio provides an integrated DBC Importer to ease and speed up the configuration of the CAN stack, if an appropriate DBC file is provided. To import LIN configurations, please continue reading at Section 4.11.2.6: Importing from the LDF format.

4.11.2.5.2. Background information

The DBC Importer can read a DBC file, fill in the system model of a project, and is able to configure one or more of the following modules:

- `Com` (`Communication`)
- `PduR` (`PDU Router`)
- `CanIf` (`CAN Interface`)
- `Can` (`CAN Driver`)
- `EcuC` (`EcuC Module`)

For a selected controller of a DBC file, the DBC Importer imports messages and signals into the respective module configurations. The PDUs created are routed from CanIf to Com (receive messages) and from Com to CanIf (transmit messages) respectively.

Note

The DBC Importer can only import CAN networks from a DBC file.

DBC files provide a feature that allows (and is commonly used by) the creator of the DBC file to attach any number of additional so-called *attributes* to messages and signals. The meaning of those attributes is defined by the creator of the DBC file, so the DBC Importer can naturally not import the information defined there. There is, however, a possibility for the user to provide a meaning for those attributes by creating a *DBC Attribute Definition File* (DADF) that tells the importer what to do with certain attributes found in the DBC file. How DAD files are created is described in Section: DBC Attribute Definition Files (DADF).

During the import, the DBC Importer creates a lot of containers for messages and signals. The names chosen for these containers are based on the names defined in the DBC file (see Section 4.11.3.1.1: Name mangling algorithm for DBC and LDF Importers for details). Optionally all names can be extended by a user defined prefix and/or a calculated instance suffix.. See Section: DBC to Autosar parameter mapping for more information which names are created in which way.

Limitations

This section lists the restrictions of the EB tresos Studio DBC Importer.

- No LIN configurations can be imported with the DBC Importer. (There is an LDF Importer to do this, see Section 4.11.2.6: Importing from the LDF format)
- All messages that are either sent or received by the controller selected are expected to be signal-carrying messages. This means that all signals that are either sent or received by the controller selected will be entered into the Com configuration, the messages will be entered in all modules below. Gateways are not supported. Transport layer and Network management configuration are not supported (if there such messages in the input file, you can use the "Do not import Can Ids" field to prevent them from being inserted in the ECU configuration).
- Not every DBC file may be understood by the DBC Importer, because DBC is a proprietary format and the DBC Importer can only interpret a subset of it (which was up to now sufficient to extract the information required). See Section: DBC grammar for more information on this subject.
- The DBC Importer currently cannot create IPDUM configurations, so multiplexed messages will be imported as empty PDUs, the MUX and signal information must be added by hand.

4.11.2.5.3. Importing DBC configurations

The DBC Importer enables the import into the system model and into the ECU configuration of the project. The GUI of the importer enables entering parameters for system model as well as ECU configuration.

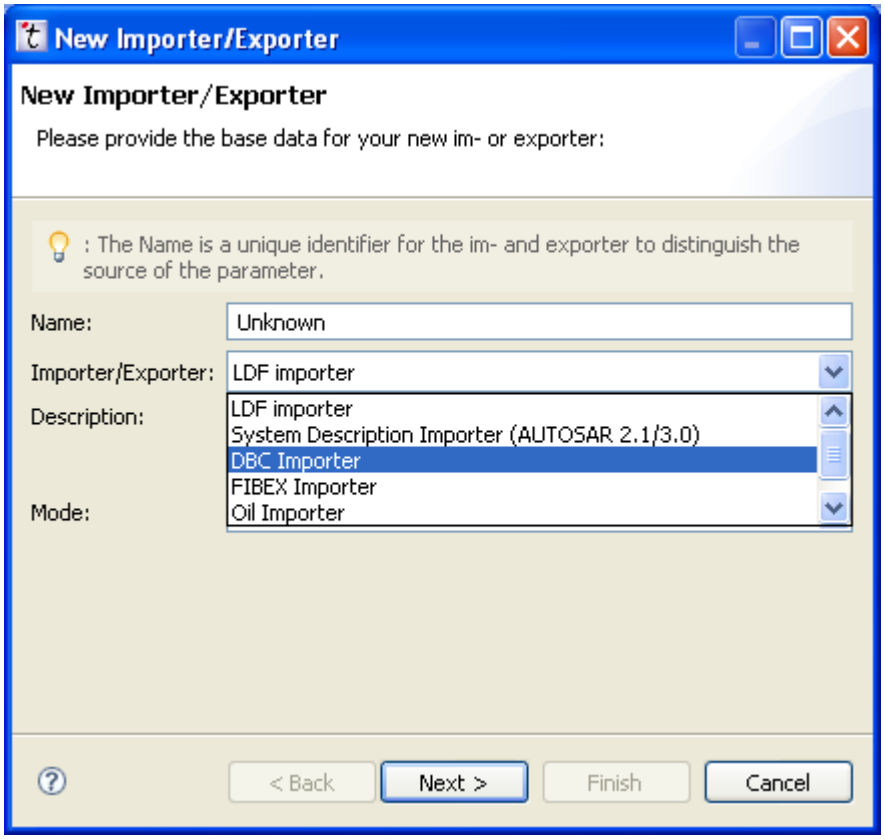
To import configurations from the DBC format:

- Refer to Section 4.11.2: Importing and exporting configuration data.
- Follow the instructions of Step 1 until you arrive at the window *Create, manage and run im- and exporters*.

In the window Create, manage and run im- and exporters:

- Select the **Create a new im- or exporter** button.

The *New Importer/Exporter* window opens up.



To enter the basic information for the new importer or exporter configuration:

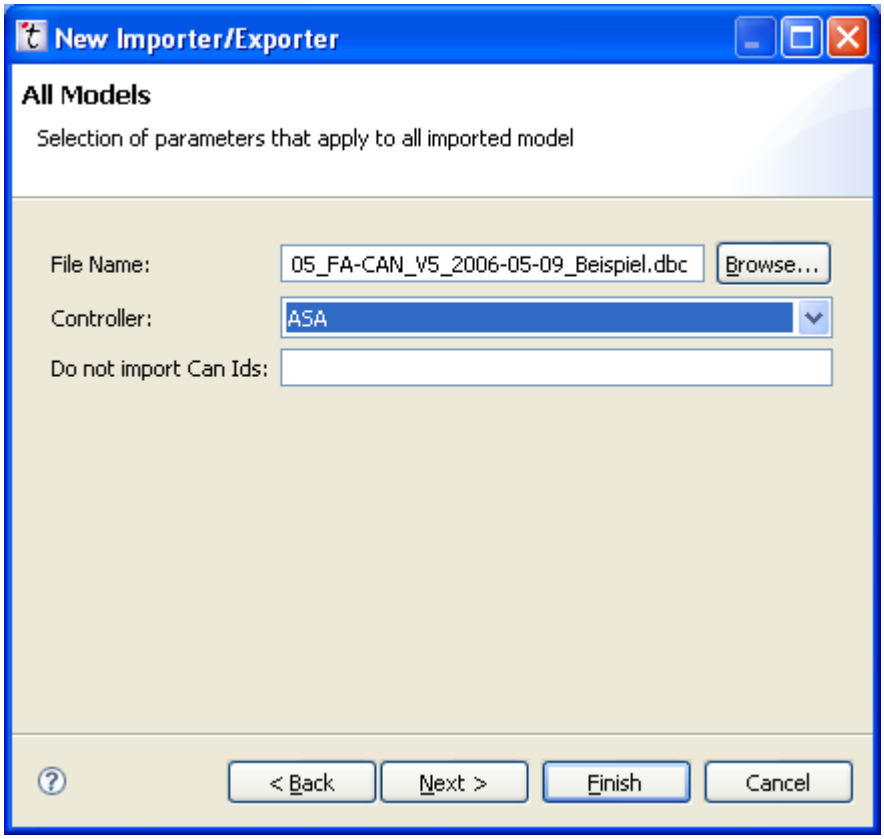
- Select the *Name* textbox.
- Enter a name for your new configuration.
- Select *DBC Importer* from the **Importer/Exporter** dropdown listbox.
- Select the *Mode* dropdown listbox.
- Select **Import only** from the dropdown listbox.

The DBC Importer can only be used to import files. Exporting to DBC format is not possible.

- Select **Next**.

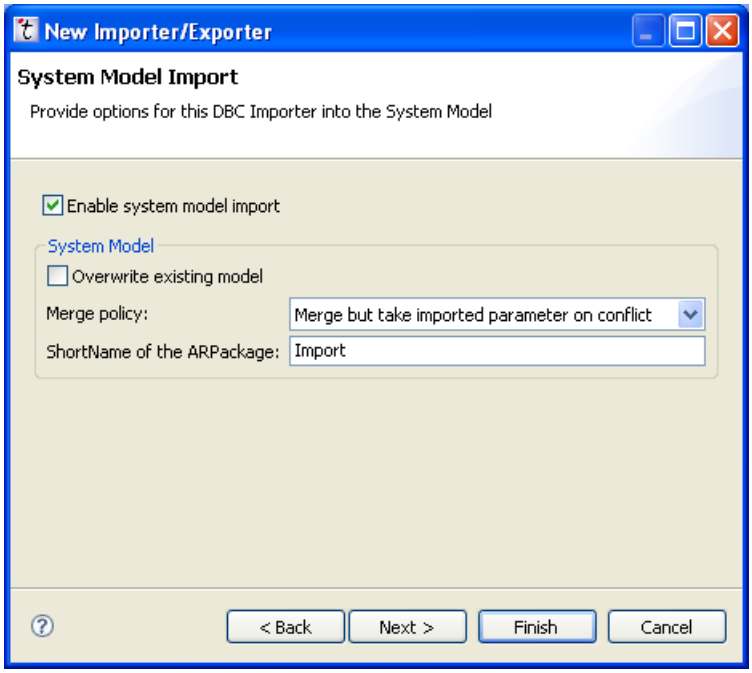
  The *All Models* window opens up.

The *All Models* window is provided for entering general information.



- Select the *File Name* file selection entry.
- Select the DBC file that you would like to import.
- In the *Controller* dropdown listbox, select the controller contained in the DBC file you would like to import. This field is populated with the controllers available after a DBC file is selected
- In the **Do not import Can-Ids** textbox, you may specify ranges of CAN-IDs that you do not want to import. You may enter multiple IDs or ID-ranges, both in decimal or hexadecimal notation separated by semicolons. For example: `0-10;0x12;0x14-0x16;` filters IDs from 0 to 10, 0x12 and 0x14 to 0x16. Syntax errors are displayed while you type. .
- Select **Next**.

  The *System Model Import* window opens up.

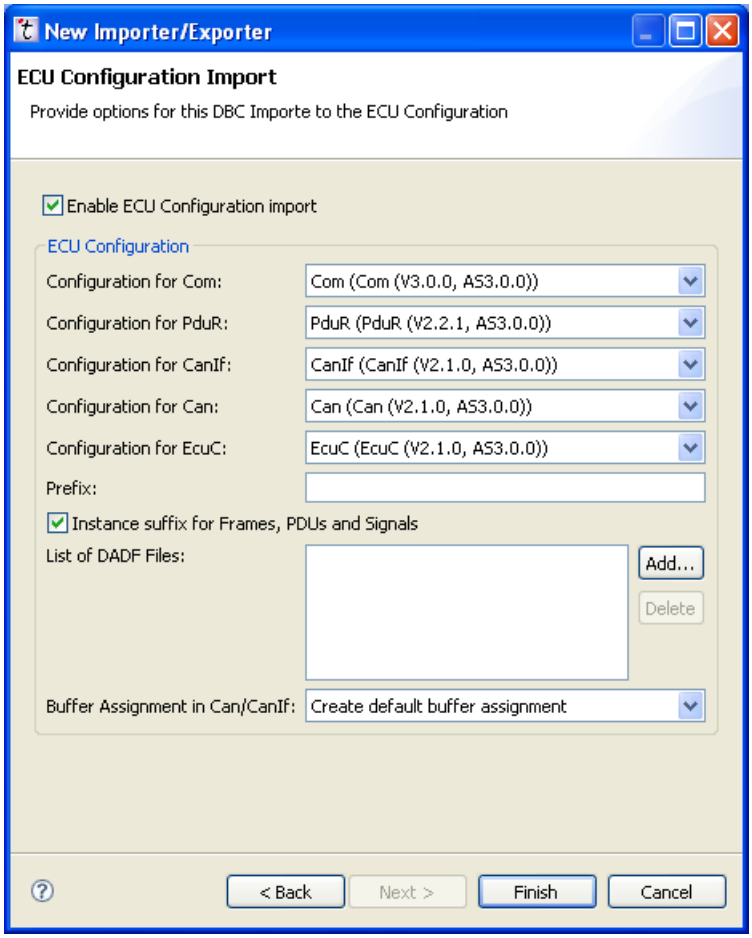


- To enable system model import into the project, check the *Enable system model import* checkbox. If the project has no system model, the complete page including the checkbox is greyed out.
- To completely replace the system model in the project, check the checkbox *Overwrite existing model*.
- To provide a hint for the merge algorithm that merges the imported configuration into the system model of the project, select an entry from the *Merge policy* dropdown listbox .
- To select the Autosar SHORT-NAME of the Autosar AR-PACKAGE into which the system model of the source file is to be imported, enter the Autosar SHORT-NAME into the **ShortName of the ARPackage** textbox.
- Select **Next** to proceed to the next configuration window.

The *ECU Configuration Import* window is provided for entering ECU configuration-specific parameters.

Note

If an ECU configuration is to be imported , you need to assure that a module configuration is present for every module you would like to import into. In the current example, these are: `Com`, `PduR`, `CanIf`, `Can`, and `EcuC`.



- To enable ECU configuration import, check the *Enable ECU Configuration Import* checkbox.
- In the *Configuration for Com* dropdown listbox, select the target Com configuration for the import.
- In the *Configuration for PduR* dropdown listbox, select the target PduR configuration for the import.
- In the *Configuration for CanIf* dropdown listbox, select the target CanIf configuration for the import.
- In the *Configuration for Can* dropdown listbox, select the target Can configuration for the import.
- In the *Configuration for EcuC* dropdown listbox, select the target EcuC configuration for the import.
- For details on the *Prefix* textbox, see Section 4.11.3.1.2: Name prefixing.
- For details on the *Instance Handling* checkbox see Section 4.11.3.1.3: Instance handling.
- In the *List of DADF Files* listbox, you may optionally add one or more DADF files to be applied during the import.

To configure the Buffer Assignment that shall be made during DBC import:

- In the *Buffer Assignment in Can/CanIf* dropdown listbox you can select the type of buffer assignment that shall be made during import of Can and CanIf. Currently two options are available.

- *Create default buffer assignment*: Up to four hardware objects (send and/or receive hardware objects for Basic and/or Extended CanIds) are configured.
- *Do not create buffer assignment*: No hardware object is created; they must be configured manually.

To save the importer configuration:

- Select **Finish**.

This saves your importer configuration and closes the window.

- To run the DBC Importer, select the importer name you just created on the left side of the *Create, manage, and run im- and exporters* window.
- Select **Run Importer**.

The DBC configuration is being imported.

4.11.2.5.4. References

DBC to Autosar parameter mapping

The DBC Importer of EB tresos Studio creates configurations for the Autosar modules Can, CanIf, PduR, Com, and EcuC. The chapter Section: DBC Importer naming convention for container prefixes lists and describes the containers and parameters created and on which DBC parameters and attributes they are based.

Subsections are named by module names or container names respectively. The symbol `<PREFIX>` represents the prefix provided by the user for the import. The symbol `<INSTSUFFIX>` represents the optional, calculated instance suffix for frames, signals, and PDUs.

DBC Importer naming convention for container prefixes

The following section lists the name extensions the DBC Importer uses when it creates containers.

- `<name>` represents the name of the signal or the message respectively as defined in the DBC file.
- `<ctrlName>` represents the name of the controller selected for import.
- Names may be mangled, for details see Section 4.11.3.1: Naming rules.

Autosar container	Name format
`CanController`	`<PREFIX><ctrlName>`
`CanIfControllerConfig`	`<PREFIX><ctrlName>`

Autosar container	Name format
CanIfDriverConfig	<PREFIX>CanIfDriverConfig
CanIfNetworkConfig	<PREFIX>CanNetwork
CanIfRxPduConfig	<PREFIX><name><INSTSUFFIX>
CanIfTxPduConfig	<PREFIX><name><INSTSUFFIX>
PduRRoutingPath	<PREFIX><name><INSTSUFFIX>
ComIPdu	<PREFIX><name><INSTSUFFIX>
ComIPduGroup	▶ ImportedRxPdus for Autosar 2.1 configurations, additionally CanRxPduGroup for Autosar 3.0 configurations ▶ ImportedTxPdus for Autosar 2.1 configurations, additionally CanTxPduGroup for Autosar 3.0 configurations
ComNetworkSignal	NW<PREFIX><name><INSTSUFFIX>
ComSignal	SG<PREFIX><name><INSTSUFFIX>
Pdu (EcuC module)	<PREFIX><name><INSTSUFFIX>

The following sections outline how the DBC Importer performs the parameter mapping for Autosar 2.1 and Autosar 3.0 modules.

Autosar 2.1 Can module

The DBC Importer creates a basic CAN driver configuration without hardware-specific details.

CanController	A CanConfigSet/CanController container is created with the name <PREFIX><name>, where <name> is the name of the controller selected. For this controller two filter masks are created: AcceptAllStd and AcceptAllExt. These are set to the filter values necessary to accept all standard and extended identifiers, which can either be 0x7ff and 0x1fffffff if the can driver implementation assumes a set bit as don't care bit value, or 0x0 for both standard and extended identifiers if the don't care bit value is a cleared bit.
CanHardwareObject	If the *Buffer Assignment in Can/CanIf* parameter has been set to *Create default buffer assignment*, the DBC Importer creates up to four hardware objects in the container CanConfigSet: ▶ a transmit object (*Hardware Transmit Handle* , *HTH*) for messages with standard identifiers, ▶ a transmit object for messages with extended identifiers,

	▶ a receive object (*Hardware Receive Handle* , *HRH*) for messages with standard identifiers, and ▶ a receive object for messages with extended identifiers. The names of these hardware objects are: ▶ `TxStd<PREFIX><Controllername>` (HthStandard) ▶ `TxExt<PREFIX><Controllername>` (HthExtended) ▶ `RxStd<PREFIX><Controllername>` (HrhStandard) ▶ `RxExt<PREFIX><Controllername>` (HrhExtended) Each of them is created only if there is at least one import message that requires it. The DBC Importer automatically links these HRHs and HTHs to the controller created (see `CanIfControllerConfig`). The `CanObjectId` parameter is set in such a way that the parameter values are zero-based and dense and HrhStandard and HrhExtended objects obtain lower values than HthStandard and HthExtended objects.

Autosar 2.1 CanIf Module

The CAN Interface configuration imported by the DBC Importer contains the definition of all messages that are sent or received by the controller selected. Additionally the `CanIfDriverConfig`, `CanIfControllerConfig`, and `CanIfNetworkConfig` are manipulated.

CanIfDriverConfig	If a `CanIfDriverConfig` does not yet exist, the container is created with the name `<PREFIX>CanIfDriverConfig`. If a subcontainer already exists (e.g. the default container `CanIfDriverConfig_0`), the content of this container is configured without changing its name. If the *Buffer Assignment in Can/CanIf* parameter has been set to *Create default buffer assignment*, up to four hardware objects are created. The exact number depends on the actual configuration (similar to what is imported into `CanHardwareObject` in the CAN module, Section: Autosar 2.1 Can module). The hardware objects are configured for sending and receiving CAN messages with standard and extended identifiers. They reference the respective hardware object in the CAN Driver configuration and the `CanIfControllerConfig` created. The parameters `CanIfHthType` and `CanIfHrhType` are set to `BasicCan`.
CanIfControllerConfig	A `CanIfControllerConfig` container is created with the name `<PREFIX><name>`, where `<name>` is the name of the controller selected. The controller created references the CAN Driver controller created (see `CanController` in the Can module, Section: Autosar 2.1 Can module), the CAN Interface

	driver configuration created (see container `CanIfDriverConfig` in the CanIf module, Section: Autosar 2.1 CanIf Module), and the CAN Interface network configuration created (see container `CanIfNetworkConfig` in the CanIf module, Section: Autosar 2.1 CanIf Module).
CanIfNetworkConfig	The `CanIfNetworkConfig` container is created with the name `<PREFIX>CanNetwork`.
CanIfInitConfigSet	The container `CanIfInitConfigSet` is created. This container contains the following subcontainers. ▶ CanIfInitNetworkConfig ▶ CanIfInitControllerConfig ▶ CanIfInitDriverConfig ▶ CanIfInitConfig Below the `CanIfInitNetworkConfig` list container, one subcontainer is created with the name `<PREFIX>CanNetworkInitNW`. ▶ The parameter `CanIfControllerConfigSet` is set to `CANIF_NETWORK_0_CONFIG_0`. ▶ The link parameter `CanIfNetworkRef` is set to the enclosing container. ▶ The containers `CanIfInitControllerConfig` and `CanIfInitDriverConfig` are left untouched by the DBC Importer. ▶ If a container below `CanIfInitConfig` already exists, this container is used for inserting the Rx and Tx PDUs. ▶ If such a container does not exist, a new container with the name `CanIfInitConfig` is created. The following sections describe how the DBC Importer fills this container.
CanIfInitConfig	If a `CanIfInitConfig` entry in the `CanIfInitConfig` list does not yet exist, the container is created with the name `CanIfInitConfig`. If a subcontainer already exists, the content of this container is configured without changing its name.
CanIfRxPduConfig	For every message received by the controller, a `CanIfRxPduConfig` container is created below `CanIfInitConfigSet/CanIfInitConfig`. This container is named `<PREFIX><name><INSTSUFFIX>`, where `<name>` is the name of the message in the DBC file. The following parameters are set: `CanIfRxPduRxIndiUser` is set to `PduR`

	`CanIfRxPduRxIndiCallOutFun`
	is set to `PduR_CanIfRxIndication`
	
	`CanIfRxPduIdCanIdType`
	is set to `Standard` or `Extended` as defined in the DBC file
	
	`CanIfRxTargetPduID`
	is set identical to `CanIfCanRxPduId`
	
	`CanIfCanRxPduCanId`
	is set to the CAN Identifier as defined in the DBC file
	
	`CanIfCanRxPduDlc`
	is set to the *DLC* as defined in the DBC file
	
	Additionally the following references are set: If the *Buffer Assignment in Can/CanIf* parameter has been set to *Create default buffer assignment*, the matching hardware object of the `CanIfDriverConfig` created is referenced (see container `CanIf-DriverConfig` in the CanIf module, Section: Autosar 2.1 CanIf Module). The `CanIfNetworkConfig` created is referenced (see container `CanIfNetworkConfig` in the CanIf module, Section: Autosar 2.1 CanIf Module), as well as the global PDU container in the EcuC (see Section: Autosar 2.1 EcuC module).

CanIfTxPduConfig	For every message sent by the controller, a `CanIfTxPduConfig` container is created below `CanIfInitConfigSet/CanIfInitConfig`. This container is named `<PREFIX><name><INSTSUFFIX>`, where `<name>` is the name of the message in the DBC file.
	
	The following parameters are set:
	
	`CanIfTxUserType`
	is set to `PduR`
	
	`CanIfTxPduTxConfirmCallOut`
	is set to `PduR_CanIfTxConfirmation`
	
	`CanIfTxPduIdCanIdType`
	is set to `Standard` or `Extended` as defined in the DBC file
	
	`CanIfCanTxPduCanId`
	is set to the CAN Identifier as defined in the DBC file
	
	`CanIfCanTxPduDlc`
	is set to the DLC as defined in the DBC file
	
	If the *Buffer Assignment in Can/CanIf* parameter has been set to *Create default buffer assignment*, the matching hardware object of the `CanIfDriverConfig` created is referenced (see container `CanIfDriverConfig` in the CanIf module, Section: Autosar 2.1 CanIf Module). The `CanIfNetworkConfig` created is refer-

	enced (see container `CanIfNetworkConfig` in the CanIf module, Section: Autosar 2.1 CanIf Module), as well as the global PDU associated with this message (see Section: Autosar 2.1 EcuC module).

Autosar 2.1 PduR module

The PDU Router configuration created by the DBC Importer contains routing paths for all messages sent or received by the controller selected. All received messages are routed from the CanIf module to the Com module, and all sent messages are routed from the Com module to the CanIf module.

PduRGlobalConfig/PduRRoutingPath	The DBC Importer creates a `PduRGlobalConfig/PduRRoutingPath` container for every message sent or received by the chosen controller. This container is named after the message in the DBC file in accordance with the name mangling described in Section 4.11.3.1: Naming rules. The importer creates a `PduRSrcPdu` and a `PduRDestPdu` container for this `PduRRoutingPath`. Both the source and the destination PDU containers are set up to reference the global PDU associated with the frame (see `PduCollection/Pdu` in Section: Autosar 2.1 EcuC module). The DBC Importer sets the parameter `SduLength` to the bytelength of the SDU containing the PDU.

Autosar 2.1 Com module

The DBC Importer COM configuration contains all messages sent or received by the controller. It also contains all signals contained within these messages.

ComIPduGroup PDU groups	If not already present, two general PDU groups are created. They are called `RXPDUS_GLOBAL` and `TXPDUS_GLOBAL`. Two sub PDU groups called `RG<PREFIX>CanNetwork` and `TG<PREFIX>CanNetwork` are additionally created and assigned as subgroup to their matching general group.
ComIPdu	For every message sent or received by the controller selected a `ComIPdu` container is created. The container is named `PD<PREFIX><name><INSTSUFFIX>` , with `<name>` being the name of the message in the DBC file. Every I-PDU references its associated global PDU (see `PduCollection/Pdu` in Section: Autosar 2.1 EcuC module) and has the following parameters set: **Parameter** — **Value** `ComIPduSize` — set to the DLC as defined in the DBC multiplied by eight

<table>
<tr><td></td><td>All I-PDUs that are received by the controller selected reference the associated ComIPduGroup RG<PREFIX>CanNetwork and have the following parameters set:
<table>
<tr><th>Parameter</th><th>Value</th></tr>
<tr><td>ComIPduDirection</td><td>is set to Received</td></tr>
</table>
All I-PDUs that are sent by the controller selected reference the associated ComIPduGroup TG<PREFIX>CanNetwork and have the following parameters set:
<table>
<tr><th>Parameter</th><th>Value</th></tr>
<tr><td>ComIPduDirection</td><td>is set to Sent</td></tr>
</table></td></tr>
<tr><td>ComSignal</td><td>For every signal sent or received by the controller selected, a ComSignal container is created. The container is named SG<PREFIX><name><INSTSUFFIX>, whith <name> being the name of the signal in the DBC file.
Every ComSignal container references the associated ComNetworkSignal container and has the following parameters set:
<table>
<tr><th>Parameter</th><th>Value</th></tr>
<tr><td>ComSignalLength</td><td>is set to the number of bytes required for the signal if the parameter ComSignalType is set to UInt8_n.</td></tr>
<tr><td>ComSignalType</td><td>▶ is set to Boolean, if signal size is one bit;
▶ is set to UInt8/SInt8 for unsigned/signed signals with size less than or equal to 8;
▶ is set to UInt16/SInt16 for unsigned/signed signals with size less than or equal to 16;
▶ is set to UInt32/SInt32 for unsigned/signed signals with size less than or equal to 32;
▶ is set to UInt8_n for signals with size greater than 32</td></tr>
<tr><td>ComSignalProperty</td><td>▶ is set to SendStaticExternal for signals sent
▶ ReceiveUnqueuedExternal for signals received</td></tr>
<tr><td>ComSignalSystemSignalShortName</td><td>This vendor-specific parameter is set to the original name of the signal as defined in the DBC file.</td></tr>
</table></td></tr>
<tr><td>ComNetworkSignal</td><td>For every signal sent or received by the controller a ComNetworkSignal container is created. The container is named NW<PREFIX><name><INSTSUFFIX>, with</td></tr>
</table>

	<name> being the name of the signal in the DBC file. Every ComNetworkSignal container references the ComIPdu container of the PDU the signal is assigned to and has the following parameters set: **Parameter** / **Value** ComNetworkSignalSize: is set to the signal length from the DBC ComNetworkSignalBitPosition: is set to the start bit as defined in the DBC ComNetworkSignalEndianess: ▶ is set to LittleEndian or BigEndian as defined in the DBC (called *Intel* and *Motorola* there); ▶ for byte-array signals it is set to Opaque ComNetworkSignalDirection: ▶ is set to Sent for signals sent ▶ is set to Received for signals received

Autosar 2.1 EcuC module

The EcuC configuration created by the DBC Importer contains a global PDU for all messages that are sent or received by the selected controller.

PduCollection/Pdu	The DBC Importer creates a PduCollection/Pdu container named <PREFIX><name><INSTSUFFIX>, where <name> is the name of the frame in the DBC file.

Autosar 3.0 Can module

The DBC Importer handles Autosar 3.0 Can modules in essentially the same way as Autosar 2.1 Can modules. Refer to Section: Autosar 2.1 Can module for details.

Autosar 3.0 CanIf Module

The CAN Interface configuration imported by the DBC Importer contains the definition of all messages that are sent or received by the controller selected. Additionally the CanIfDriverConfig and CanIfControllerConfig, containers are manipulated.

CanIfDriverConfig	If a `CanIfDriverConfig` does not yet exist, the container is created with the name `<PREFIX>CanIfDriverConfig`. If a subcontainer already exists (e.g. the default container `CanIfDriverConfig_0`), the content of this container is configured without changing its name. In this container, the associated `CanIfInitHohConfig` container and the `CanGeneral` container of the corresponding Can configuration are referenced, see Section: Autosar 3.0 Can module;
CanIfControllerConfig	A `CanIfControllerConfig` container is created with the name `<PREFIX><name>`, where `<name>` is the name of the controller selected. The controller created references the CAN Driver controller created (see `CanController` in the Can module, Section: Autosar 3.0 Can module), the CAN Interface driver configuration created (see container `CanIfDriverConfig` in the CanIf module, Section: Autosar 3.0 CanIf Module), and the corresponding `CanIfInitControllerConfig` container, see Section: Autosar 3.0 CanIf Module).
CanIfInitConfiguration	If a `CanIfInitConfiguration` container does not yet exist, it is created with the name `CanIfInitConfiguration`. If the container already exists (e.g. the default container `CanIf_InitConfig_0`, the content of this container is configured without changing its name. The following subcontainers are created: ▶ CanIfInitHohConfig ▶ CanIfInitRxPduConfig ▶ CanIfInitTxPduConfig ▶ CanIfInitControllerConfig The following sections describe how the DBC Importer fills these containers.
CanIfInitHohConfig	If the *Buffer Assignment in Can/CanIf* parameter has been set to *Create default buffer assignment*, up to four hardware objects are created. The exact number depends on the actual configuration (similar to what is imported into `CanHardwareObject` in the CAN module, Section: Autosar 3.0 Can module). The hardware objects are configured for sending and receiving CAN messages with standard and extended identifiers. They reference the respective hardware object in the CAN Driver configuration and the `CanIfControllerConfig` created. The parameters `CanIfHthType` and `CanIfHrhType` are set to `BASIC_CAN`.
CanIfRxPduConfig	For every message received by the controller, a `CanIfRxPduConfig` container is created. This container is named `<PREFIX><name><INSTSUFFIX>`, where `<name>` is the name of the message in the DBC file. The following parameters are set:

	`CanIfRxUserType`
	is set to `PDUR`
	
	`CanIfUserRxIndication`
	is set to `PduR_CanIfRxIndication`
	
	`CanIfRxPduIdCanIdType`
	is set to `STANDARD_CAN` or `EXTENDED_CAN` as defined in the DBC file
	
	`CanIfCanRxPduCanId`
	is set to the CAN Identifier as defined in the DBC file
	
	`CanIfCanRxPduDlc`
	is set to the *DLC* as defined in the DBC file
	
	Additionally the following references are set: If the *Buffer Assignment in Can/CanIf* parameter has been set to *Create default buffer assignment*, the matching hardware object is referenced (see container `CanIfInitHohConfig` in the CanIf module, Section: Autosar 3.0 CanIf Module). A link to the global PDU container in the EcuC (see Section: Autosar 3.0 EcuC module) is also set.

CanIfTxPduConfig	For every message sent by the controller, a `CanIfTxPduConfig` container is created. This container is named `<PREFIX><name><INSTSUFFIX>`, where `<name>` is the name of the message in the DBC file.
	
	The following parameters are set:
	
	`CanIfTxUserType`
	is set to `PDUR`
	
	`CanIfUserTxConfirmation`
	is set to `PduR_CanIfTxConfirmation`
	
	`CanIfTxPduIdCanIdType`
	is set to `STANDARD_CAN` or `EXTENDED_CAN` as defined in the DBC file
	
	`CanIfCanTxPduIdCanId`
	is set to the CAN Identifier as defined in the DBC file
	
	`CanIfCanTxPduDlc`
	is set to the DLC as defined in the DBC file
	
	Additionally the following references are set: If the *Buffer Assignment in Can/CanIf* parameter has been set to *Create default buffer assignment*, the matching hardware object is referenced (see container `CanIfInitHohConfig` in the CanIf module, Section: Autosar 3.0 CanIf Module). A link to the global PDU container in the EcuC (see Section: Autosar 3.0 EcuC module) is also set.

Autosar 3.0 PduR module

The DBC Importer handles Autosar 3.0 PduR modules in essentially the same way as Autosar 2.1 PduR modules. Refer to Section: Autosar 2.1 PduR module for details.

Autosar 3.0 Com module

The DBC Importer COM configuration contains all messages sent or received by the controller. It also contains all signals contained within these messages.

<table>
<tr><td>ComIPduGroup PDU groups</td><td>If not already present, two general PDU groups are created. They are called RXPDUS_GLOBAL and TXPDUS_GLOBAL. Two sub PDU groups called RG<PREFIX>CanNetwork and TG<PREFIX>CanNetwork are additionally created and assigned as subgroup to their matching general group.</td></tr>
<tr><td>ComIPdu</td><td>For every message sent or received by the controller selected a ComIPdu container is created. The container is named PD<PREFIX><name><INSTSUFFIX> , with <name> being the name of the message in the DBC file.

Every I-PDU references its associated global PDU (see PduCollection/Pdu in Section: Autosar 3.0 EcuC module) and has the following parameters set:

<table><tr><th>Parameter</th><th>Value</th></tr><tr><td>ComIPduSize</td><td>set to the DLC as defined in the DBC</td></tr><tr><td>ComIPduSignalRef</td><td>For every signal that is contained in the Ecu which the current controller actually receives or sends, one reference to the corresponding ComSignal container is added to this reference list.</td></tr></table>
All I-PDUs that are received by the controller selected reference the associated ComIPduGroup RG<PREFIX>CanNetwork and have the following parameters set:

<table><tr><th>Parameter</th><th>Value</th></tr><tr><td>ComIPduDirection</td><td>is set to RECEIVE</td></tr></table>
All I-PDUs that are sent by the controller selected reference the associated ComIPduGroup TG<PREFIX>CanNetwork and have the following parameters set:

<table><tr><th>Parameter</th><th>Value</th></tr><tr><td>ComIPduDirection</td><td>is set to SEND</td></tr></table></td></tr>
</table>

ComSignal	For every signal sent or received by the controller selected, a `ComSignal` container is created. The container is named `SG<PREFIX><name><INSTSUFFIX>`, whith `<name>` being the name of the signal in the DBC file. Every `ComSignal` has the following parameters set:

Parameter	Value
`ComSignalLength`	is set to the number of bytes required for the signal if the parameter `ComSignalType` is set to `UINT8_N`.
`ComSignalType`	▶ is set to `BOOLEAN`, if signal size is one bit; ▶ is set to `UINT8/SINT8` for unsigned/signed signals with size less than or equal to 8; ▶ is set to `UINT16/SINT16` for unsigned/signed signals with size less than or equal to 16; ▶ is set to `UINT32/SINT32` for unsigned/signed signals with size less than or equal to 32; ▶ is set to `UINT8_N` for signals with size greater than 32
`ComBitSize`	is set to the signal length from the DBC
`ComBitPosition`	is set to the start bit as defined in the DBC
`ComSignalEndianess`	▶ is set to `LITTLE_ENDIAN` or `BIG_ENDIAN` as defined in the DBC (called *Intel* and *Motorola* there); ▶ for signals with a `ComSignalType` of `UINT8_N` it is set to `OPAQUE`
`ComSignalSystemSignalShortName`	This vendor specific parameter is set to the original name of the signal as defined in the DBC file.
`SystemTemplateSystemSignalRef`	This reference parameter is set to the corresponding `ISignalToIPduMapping` container of the imported system model.

Autosar 3.0 EcuC module

The EcuC configuration created by the DBC Importer contains a global PDU for all messages that are sent or received by the selected controller.

PduCollection/Pdu	The DBC Importer creates a `PduCollection/Pdu` container named `<PREFIX><name><INSTSUFFIX>`, where `<name>` is the name of the frame in the DBC file. The parameter `PduLength` is set to the length of the frame as defined in the DBC file multiplied by eight.

DBC Attribute Definition Files (DADF)

DBC files provide a feature that allows (and is commonly used by) the creator of the DBC file to attach any number of additional so-called *attributes* to messages and signals. The meaning of those attributes is defined by the creator of the DBC file, so the DBC Importer can naturally not import the information defined there.

DAD files are xml files that can be used to define how those attributes shall be evaluated by the importer. To introduce the format, we proceed as follows. First, the possible xml-tags of a DAD file are defined, together with their allowed attributes. Then the syntax, i.e. the allowed inclusion structure of the xml-tags, is declared. Given this, we provide the semantical significance of the defined objects. Finally, we supply a short, commented example of a DAD file.

The following text:

- lists the limitations of the DADF format,
- defines the possible XML tags of a DADF and the allowed attributes of these XML tags,
- declares the syntax (the allowed inclusion) of the XML tags, and
- provides the semantic significance of the defined objects.

At the end of this information, you will find a DADF example with comments.

Limitations

This section lists the limitations of the DADF mapping.

- Only Attributes defined for signals and messages can be accessed
- There are no conditionals, i.e. if an attribute is mapped to an AUTOSAR-element identified by its `DEFINITON-REF`, this mapping will uncondionally be applied to all elements identified, for which the attribute is set.

DADF XML tags and attributes

The following table lists the XML tags and the attributes allowed:

XML tags	Attributes
assignments	--
path	name
valueMap	attrName
default	value
attribute	attrName
mapping	from, to

Explanation of the syntax definition

The XML syntax for DAD files is defined by using production rules.

- The rule `left --> right` means that `left` includes/produces `right`.
- The asterisk `*` is the Kleene star, thus `object*` means that `object` may appear any number of times, including never.
- The plus symbol `+` in `object+` means that `object` may appear any number of times but at least once.
- A vertical bar separates parameters in a list from which one parameters must be chosen or used, thus `left | right` means `left` or `right`.
- The question mark `?` in `object?` means that `object` may occur once or never.

DADF XML syntax

The following inclusion structure is allowed in DADF files:

- `assignments --> path*`
- `path --> (valueMap | default | attribute) path*`
- `valueMap --> mapping+`
- `default -->`
- `attribute -->`
- `mapping -->`

DADF XML semantics

XML tag	Semantics
`assignments`	The tag is the outermost tag and serves as parenthesis for the remaining tags.

`path`	This tag serves to specify the Autosar parameters to be inserted/modified by the DAD file by identifying its schema path. Paths start at module level, i.e. they may not contain an AR-Package name.
	Define paths in the `name` attribute of the `path` tag.
	At every point in the DAD file, there is a current path. This path is empty as long as no path has been specified. For every new open `path` tag, the content of the `name` attribute of this tag is added to the current path.
	Example: To set the path `Com/ComSignal/ComSignalTransferProperty` as the current path, use the `path` tag:
	`<path name="Com/ComSignal/ComSignalTransferProperty"></path>`
	You can also nest several `path` tags:
	`<path name="Com/ComSignal">`
	`  <path name="ComSignalTransferProperty"></path>`
	`</path>`

`valueMap`	Defines how the value for an Autosar parameter is retrieved from a DBC file. Use the tag with one or more mapping tags to map enumeration values from DBC to values in EB tresos Studio. This tag is always wrapped into one or more `path` tags. Therefore, whenever it is used, the DAD file's current path at the `valueMap` tag position is not empty. That current path needs to reference an Autosar parameter.
`default`	Defines how the value for an Autosar parameter is retrieved from a DBC file. To override the default value of a parameter, specify your own default value with the `default` tag. To do so, set the attribute `value` of the tag to the desired value. This tag is always wrapped into one or more `path` tags. Therefore, whenever it is used, the DAD file's current path at the `default` tag position is not empty. That current path needs to reference an Autosar parameter.
`attribute`	Defines how the value for an Autosar parameter is retrieved from a DBC file.

XML tag	Semantics
	If attribute A has an ordinal value, such as an integer or a string, and you want the Autosar parameter (selected by the current path) to receive that value, use the `attribute` tag and specify the name of A in `attrName`. This tag is always wrapped into one or more `path` tags. Therefore, whenever it is used, the DAD file's current path at the `attribute` tag position is not empty. That current path needs to reference an Autosar parameter.
`mapping`	Use `mapping` tags to map enumeration values from DBC to values in EB tresos Studio. Use the `from` attribute in the `mapping` tag to specify the enumeration value from the DBC file to the Autosar parameter. The Autosar parameter value is specified in the `to` attribute.

Example DAD file

The following is example code for a DBC Attribute Definition File. It shows how the EB tresos importer can be configured so that the values of the attributes are mapped correctly into your module configurations.

If you remove the comments, the file is a valid DAD file. The file is associated with a DBC file that defines a CAN network with just two controllers with two messages, each of which contains just one signal. This network uses three attributes.

The definitions have to be always wrapped by *assignment* tags.

```
<assignments>
```

This example shows how to map enumeration attributes of DBC to Autosar parameter values. The path given below states the path to the parameter definition targeted. In this case it states that the destination are all `ComSignal` containers of the `Com` plugin.

```
  <path name="Com/ComSignal">
```

The path given below defines the name of a parameter of the container that is specified in the above path.

```
    <path name="ComSignalTransferProperty">
```

At this point, a concrete parameter is identified. What follows is the definition of where in the DBC file the information for the value of this parameter can be found. Since the parameter specifies a property of a signal, it is assumed, that this value can be found in an attribute that is attached to a signal in the DBC file. In this

case, this is the attribute with the identifier "Attribute2". This DBC attribute is defined to be an enumeration (in the DBC file), therefore the importer needs to know which enumeration values from the DBC file are mapped to which Autosar values. This is done inside the mapping tags, where the DBC value is stored in `from`, whereas the Autosar value is stored in `to`.

```
      <valueMap attrName="Attribute2">
        <mapping from="one" to="Triggered"/>
        <mapping from="two" to="Pending"/>
        <mapping from="three" to="Triggered"/>
        <mapping from="four" to="Triggered"/>
      </valueMap>
    </path>
```

The path below defines the name of another parameter. The complete path is `Com/ComSignal/ComSignalInitValue`.

```
    <path name="ComSignalInitValue">
```

The attribute with the identifier `Attribute3` in the DBC file is of ordinal type, so here it suffices to provide the identifier of that attribute. The value of the attribute will be copied to the parameter specified by the path above.

```
      <attribute attrName="Attribute3"/>
    </path>
  </path>
```

The two closing *path* tags above remove first the trailing `ComSignalInitValue` and then the trailing `Com/ComSignal` from the current path. Note that these were the strings added to the current path by the corresponding opening *path* tags. So, at this point the current path is empty again. Therefore, in the next *path* tag, we have to start at module level again in order to specify a correct path to an Autosar parameter.

```
  <path name="Com/ComIPdu/ComIPduMinimumDelayFactor">
    <attribute attrName="Attribute1"/>
  </path>
```

All parameters that are not explicitly created by the DBC Importer will be set to their default value (see Section: DBC to Autosar parameter mapping for a list of all parameters set). DAD files, however, allow you to override that default value as shown below.

```
  <path name="PduR/PduRRoutingTable/PduRRoutingPath">
    <path name="SduLength">
```

```
      <default value="8"/>
    </path>
  </path>
</assignments>
```

The DBC parser

DBC is not an open file format, but a prorietary format defined by Vector Informatik. For this reason, not every DBC file may be importable by the DBC Importer. The following chapter describes which DBC files the DBC Importer can read.

For reading DBC files a DBC parser is applied. Every parser is based on a *grammar*. Since the grammar of the DBC format is not freely available, EB deduced a subset of their own, which allows parsing the information needed by the DBC Importer.

DBC grammar

The DBC Importer uses the following grammar to parse DBC files. The DBC Importer ignores lines that start with tags, which are not specified here.

```
dbc --> version ns bs controllers message*
        [ comment | attrdef | attrinit | attribute | envVarDef ]*

version --> "VERSION" quotedString

ns --> "NS_" ":" [ UC | "_" ]*

bs --> "BS_" ":"

controllers --> "BU_" ":" identifier+ {controller names}

message --> "BO_" integer {CAN-id} identifier {message name} ":"
            integer {size of data in bytes} identifier {sender} signal+

signal --> "SG_" identifier {signal name} ["M"|["m" integer]]? ":"
           integer {startbit}  "|" integer {bit length}  "@"
           ["0" | "1"] {endianess} ["+" | "-"] {type}
           "(" integer {factor}  ","  integer {offset}  ")"
```

```
          "[" integer {minimum}  "|" integer {maximum} "]"
          quotedString {unit [km, h, kg, ...]}
          identifier [ "," identifier]* {receiver}

comment --> "CM_" [ message-comment | signal-comment | envVar-comment |
                    controller-comment | net-Comment | otherComment ] ";"

message-comment --> "BO_" integer {CAN-id} quotedString {comment}

signal-comment --> "SG_" integer {CAN-id} identifier {signal name}
                   quotedString {comment}

envVar-comment --> "EV_" identifier quotedString {comment}

controller-comment --> "BU_" identifier quotedString {comment}

net-comment --> quotedString {comment}

otherComment -->  { informal: This rule should eat all comment for
   objects which are not supported }



attrdef --> "BA_DEF_" ["BO_" | "BU_" | "SG_" | "EV_" ]?
        quotedString {attribute name} dataType ";"

dataType --> ["STRING" |
              ["ENUM" quotedString ["," quotedString]* ] |
              ["INT" integer {lower bound} integer {upper bound} ] |
              ["HEX" integer {lower bound} integer {upper bound} ] |
              ["FLOAT" float {lower bound} float   {upper bound} ]

attrinit --> "BA_DEF_DEF_" quotedString {attribute name} attrVal ";"

attrVal --> [float | integer | quotedString]

attribute --> "BA_" quotedString {attribute name} attrDest attrVal ";"

attrDest --> [ ["BU_" identifier {controller name} ] |
               [ "BO_" integer {CAN-id} ] |
               [ "SG_" integer {CAN-id} identifier {signal name}]
               [ "EV_" identifier {env. var. name}] ]?
```

```
envVarDef --> "EV_" identifier {variable name} ":"
              integer {value type}
              "[" integer {minimum} "|" integer {maximum} "]"
              quotedString {unit}  integer {default value}
              integer {number of envVarDef}
              identifier {contains access information}
              identifier ["," identifier]* {controllers} ";"


float --> integer "." integer
integer --> [0-9]+
identifier --> [a-zA-Z][A-Z0-9_]*
nl --> new line
UC --> any upper-case character
any --> any character except double quotes (")
ws --> whitespace, i.e. space and tab
quotedString --> any character sequence enclosed in double quotes (")
```

The code uses the following conventions:

`left --> right`
: `left` is composed of `right`

`A*`
: `A` may be present zero or more times

`A+`
: `A` may be present one or more times

`A?`
: `A` may be present zero or one times

`A | B`
: either `A` or `B` must be present

quotes `""`
: denote literals i.e. strings that need to appear in this form in the DBC file

brackets `[]`
: are used to group other expressions (like parentheses do in maths)

braces `{}`
: are not part of the grammar but comments to describe how certain elements are interpreted

4.11.2.6. Importing from the LDF format

EB tresos Studio provides an integrated LDF Importer to import LIN configurations from LDF files. If an LDF file is available for import, the LDF Importer simplifies and speeds up the configuration of the LIN stack.

4.11.2.6.1. Overview

The LDF Importer can read an LDF file and is able to import into the system model of the project and to configure one or more of the following modules:

- Com (Communication)
- PduR (PDU Router)
- LinIf (LIN Interface)
- Lin (LIN Driver)
- EcuC (Pseudo module that contains the global PDU list)

If an ECU configuration is imported, the LDF Importer imports frames and signals into the respective module configurations. The PDUs created are routed from LinIf to Com (receive frames) and from Com to LinIf (transmit frames) respectively.

During the import, the LDF Importer creates many containers for messages and signals. The names chosen for these containers are based on the names defined in the LDF file. Optionally all names can be extended by a user defined prefix and/or a calculated instance suffix. See Section: LDF to Autosar parameter mapping for more information which names are created in which way.

4.11.2.6.2. LDF Importer and file format background knowledge

LDF (Lin Description File) is a standardized and open file format for LIN networks. LDF files describe the nodes connected to the LIN network, frames and signals transmitted on the network, and Schedule Tables describing the transmission pattern. A LIN network contains one master node and one or more Slave nodes. The master node is the only node for which configuration data is imported since Slave nodes are not considered to be Autosar nodes.

A specification on the file format and on the LIN protocol itself can be obtained from http://www.lin-subbus.org.

4.11.2.6.3. Importing LDF configurations

The LDF Importer allows to import into system model and into the ECU configuration of the project. The GUI of the importer allows to enter parameters for both import scenarios.

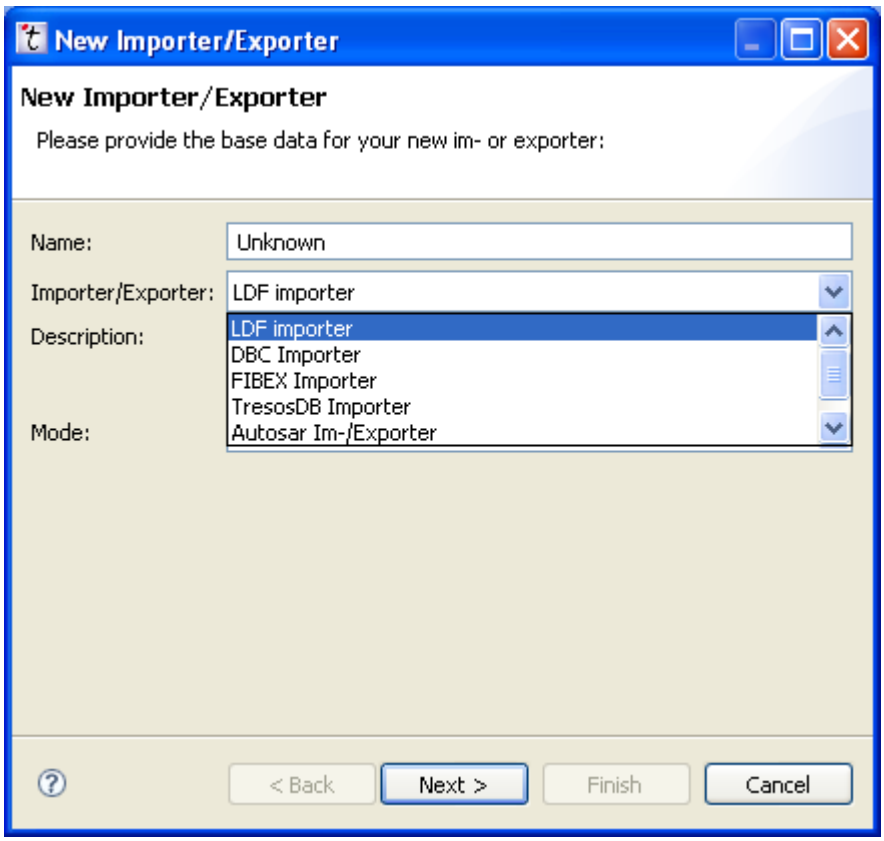
To import configurations from the LDF format:

- Refer to chapter Section 4.11.2: Importing and exporting configuration data.
- Follow the instructions of Step 1 until you arrive at the window *Create, manage and run im- and exporters*.

In the window *Create, manage and run im- and exporters*:

- Select the **Create a new im- or exporter** button.

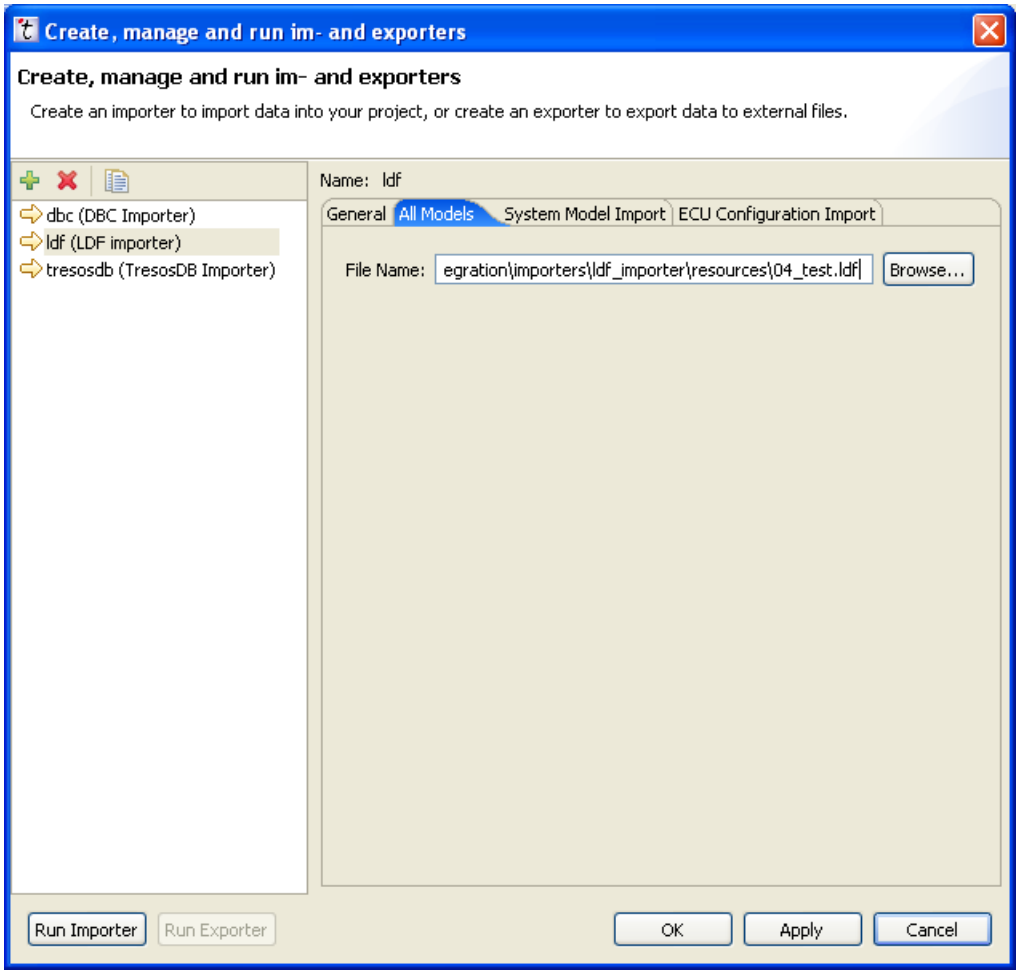
The *New Importer/Exporter* window opens up.



To enter the basic information for the new importer or exporter configuration:

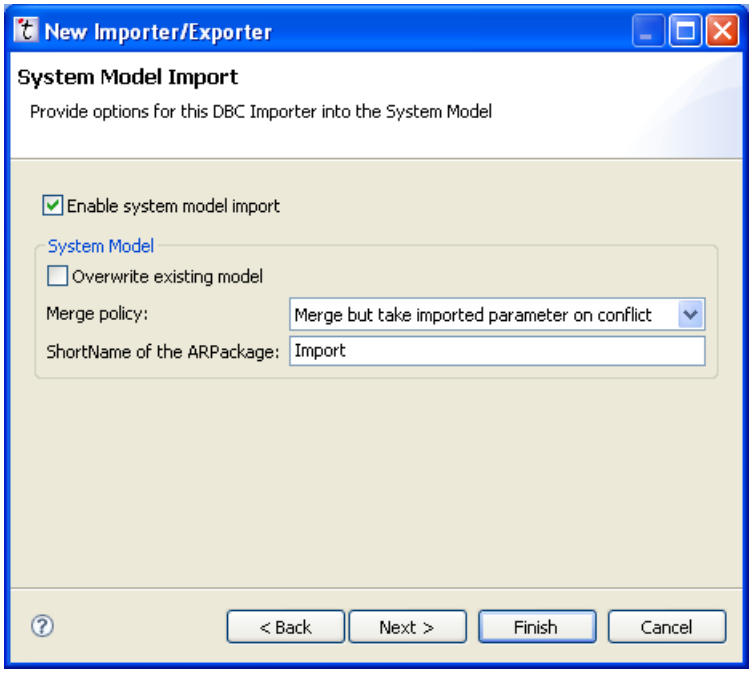
- Select the *Name* textbox.
- Enter a name for your new configuration.
- Select *LDF Importer* from the **Importer/Exporter** dropdown listbox.
- Select the *Mode* dropdown listbox.
- Select **Import only** from the dropdown listbox.
- Select **Next**.

After pressing the **Next** button you enter a page, in which you may enter general parameters.



► In the *File Name* file selection entry enter the name of an LDF file.

Press the **Next** button to switch to a page, in which you may enter parameters for the system model import.

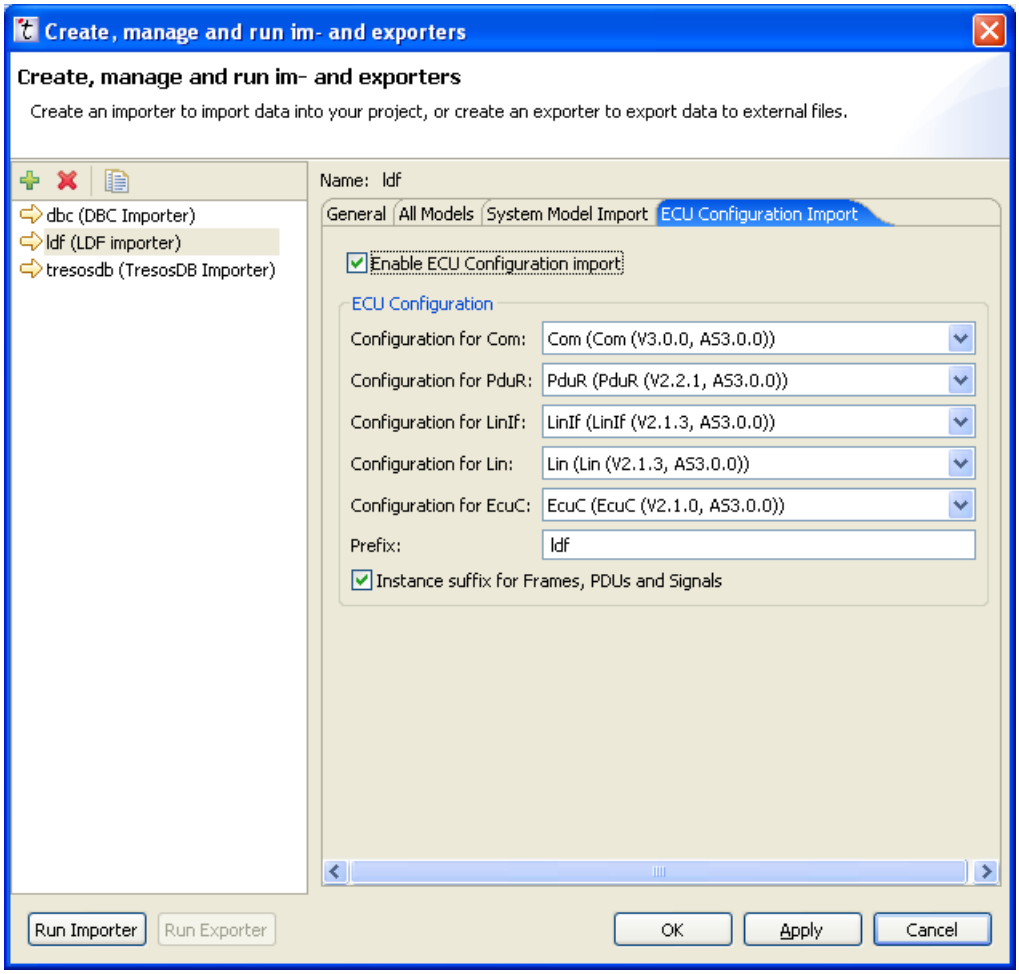


- To enable system model import into the project, check the *Enable system model import* checkbox. If the project has no system model, the complete page including the checkbox is greyed out.
- To completely replace the system model in the project, check the checkbox *Overwrite existing model*.
- To provide a hint for the merge algorithm that merges the imported configuration into the system model of the project, select an entry from the *Merge policy* dropdown listbox .
- To select the Autosar SHORT-NAME of the Autosar AR-PACKAGE into which the system model of the source file is to be imported, enter the Autosar SHORT-NAME into the **ShortName of the ARPackage** textbox.
- Select **Next** to proceed to the next configuration window.

The *ECU Configuration Import* window is provided for entering ECU configuration-specific parameters.

Note

Since the LDF Importer does not create module configurations, you need to assure that a module configuration is present for every module you would like to import into (`Com`, `PduR`, `LinIf`, `Lin`, `EcuC`).



- Check the *Enable ECU Configuration import* checkbox to enable importing ECU configuration parameters.

  Uncheck the *Enable ECU Configuration import* checkbox to disable importing ECU configuration parameters.
- In the *Configuration for Com* dropdown listbox, select the target Com configuration for the import.
- In the *Configuration for PduR* dropdown listbox, select the target PduR configuration for the import.
- In the *Configuration for LinIf* dropdown listbox, select the target LinIf configuration for the import.
- In the *Configuration for Lin* dropdown listbox, select the target Lin configuration for the import.
- In the *Configuration for EcuC* dropdown listbox, select the target EcuC configuration for the import.
- For details on the *Prefix* textbox see Section 4.11.3.1.2: Name prefixing.
- For details on the *Instance Handling* checkbox see Section 4.11.3.1.3: Instance handling.

To save the importer configuration:

- Select **Finish**.

  This saves your importer configuration and closes the *LDF Importer Options* window.

To run the LDF Importer, select the importer name you just created on the left side of the *Create, manage, and run im- and exporters* window.

- Select **Run Importer**.

  The LDF configuration is being imported.

4.11.2.6.4. References

LDF to Autosar parameter mapping

Overview

The LDF Importer of EB tresos Studio creates configurations for the Autosar modules `Lin`, `LinIf`, `PduR` , `Com`, and `EcuC`. The following section describes, which parameters are created and on which LDF parameters and attributes they are based.

Background information

Subsections are named by module names or container names respectively. The symbol `<PREFIX>` represents the prefix provided by the user for the import. The symbol `<INSTSUFFIX>` represents the optional, calculated instance suffix for frames, signals, and PDUs.

LDF Importer naming convention for container prefixes

The LDF Importer imports frames and signals into the respective module configurations. The receive PDUs created are routed from the LinIf module to the Com module and the transmit PDUs are routed from the Com module to the Lin module in turn.

The LDF Importer creates containers for messages and signals during the import. It names these containers based on the name defined in the LDF file. Optionally you may extend all names with a prefix of your own choice and/or a calculated instance suffix.

The following list is an overview of the name extensions the LDF Importer creates.

- `<name>` represents name of the signal, frame or schedule table respectively as defined in the LDF file.
- Names may be mangled, for details see Section 4.11.3.1: Naming rules.

Autosar container	Name format
LinChannel	<PREFIX>Cluster
LinIfChannel	<PREFIX>Cluster
LinIfFrame	<PREFIX><name><INSTSUFFIX>
LinIfScheduleTable	<PREFIX><name>
PduRRoutingPath	<PREFIX><name><INSTSUFFIX>
ComIPdu	<PREFIX><name><INSTSUFFIX>
ComNetworkSignal	NW<PREFIX><name><INSTSUFFIX>
ComSignal	SG<PREFIX><name><INSTSUFFIX>
Pdu (Module EcuC)	<PREFIX><name><INSTSUFFIX>

Autosar 2.1 Lin module

The LDF Importer creates a basic LIN Driver configuration.

LinGlobalConfig/LinChannel	One LinChannel container is created with the name <PREFIX>Cluster. The LDF Importer sets the following parameters: ▶ LinChannelBaudRate: is set to the baudrate of the LIN cluster as defined in the LDF file

Autosar 2.1 LinIf module

The LDF Importer imports all frames and schedule tables found in the LDF file into the LinIf module configuration.

LinIfGlobalConfig/LinIfChannel	One LinIfChannel container with the naming format <PREFIX><Cluster>is created. The importer sets the following parameters: ▶ LinIfBitrate: is set to the bitrate of the LIN cluster as defined in the LDF file. ▶ LinIfTimeBase: is set to the time base of the LIN master node as defined in the LDF file. Additionally the corresponding LIN Driver channel is referenced in LinIfChannelRef.

LinIfChannel/LinIfFrame	For every unconditional, sporadic, and event-triggered frame, a LinFrame container is created with the naming format <PREFIX><name><INSTSUFFIX>. <name> is the name of the frame in the LDF file. The Importer sets the following parameters: ► LinIfChecksumType: is set to Enhanced, if the sender and all receivers are Lin 2.x nodes (i.e. nodes conforming to LIN specification 2.0 or higher), otherwise to Classic. For sporadic and event-triggered frames, the checksum type of one of the substituted frames is set. ► LinIfFrameType: is set to Unconditional, Sporadic or EventTriggered respectively, depending on the frame definition in the LDF file. ► LinIfLength: is set to the length of the frame as defined in the LDF file. For sporadic and event-triggered frames, the frame length of one of the substituted frames is set. ► LinIfPid: is set to the protected identifier of the frame derived by the identifier provided in the LDF file. For sporadic frames, this parameter is not set. ► LinIfFixedFrameSdu: only set for MasterRequestFrames and SlaveResponseFrames. Describes a complete LIN frame (8 bytes) as part of a configuration. Additionally, the LinIfPduDirection choice container is set to LinIfTxPdu for outgoing frames and to LinIfRxPdu for incoming frames. For unconditional frames a reference to the global PDU in the EcuC module is set. Outgoing frames provide this reference set in LinIfTxPduRef, incoming frames in LinIfRxPduRef.
LinIfChannel/LinIfScheduleTable	For every schedule table defined in the LDF file, a LinIfScheduleTable container is created with the naming format <PREFIX><name>. <name> is the name of the schedule table in the LDF file. For every schedule entry, a LinIfEntry container is created. The importer sets the following parameters: ► LinIfFrameRef: is set to reference the frame, which is transmitted within the entry, as defined in the LDF file. ► LinIfDelay: is set to the duration of the frame slot as defined in the LDF file.

Autosar 2.1 PduR module

The PDU Router configuration created by the LDF Importer contains routing paths for all frames that are sent or received by the master node. All frames received are routed from *LinIf* to *Com*, all frames sent are routed from *Com* to *LinIf*.

PduRRoutingPath	For every unconditional frame sent or received by the LIN master node, a PduRGlobalConfig with the naming format <PREFIX><name><INSTSUFFIX>. <name> is the name of the frame in For this PduRRoutingPath a PduRSrcPdu and PduRDestPdu container is created. Both the s are set up to reference the global PDU associated with the frame (see container PduCollecti Autosar 2.1 EcuC module). The LDF Importer sets the parameter SduLength to the byte length of the unconditional frame

Autosar 2.1 Com module

The COM configuration created by the LDF Importer essentially contains all unconditional frames that are sent or received by the master node as well as all signals that are contained in these frames.

ComIPduGroup	If they are not already present, the LDF Importer creates two PDU groups named: ▶ ImportedRxPdus and ▶ ImportedTxPdus.
ComIPdu	For every unconditional frame sent or received by the master node there is a ComIPdu container created with the name PD<PREFIX><name><INSTSUFFIX>, where <name> is the name of the frame in the LDF file. Every I-PDU references the associated global PDU (see container PduCollection/Pdu in the ECUC module, Section: Autosar 2.1 EcuC module). The following parameters are set for every I-PDU: ▶ ComIPduSize is set to the size of the frame as defined in the LDF multiplied by eight All I-PDUs that are received by the master node reference the ImportedRxPdus ComIPduGroup and have the following parameters set: ▶ ComIPduDirection is set to Received All I-PDUs that are sent by the master node reference the ImportedTxPdus ComIPduGroup and have the following parameters set: ▶ ComIPduDirection is set to Sent
ComSignal	For every signal sent or received by the master node, the LDF Importer creates a ComSignal container named SG<PREFIX><name><INSTSUFFIX>, where <name> is the name of the signal in the LDF file.

	Every `ComSignal` container references the associated `ComNetworkSignal` container and has the following parameters set:
	▶ `ComSignalLength` is set to the number of bytes occupied by the signal if its `ComSignalType` is `UInt8_n`.
	▶ `ComSignalType`
	  ▶ is set to `Boolean`, if the signal size is one bit;
	  ▶ is set to `UInt8` for signals with a size of less than or equal to 8 bit;
	  ▶ is set to `UInt16` for signals with a size of more than 8 bit;
	  ▶ is set to `UInt8_n` for byte array signals
	▶ `ComSignalProperty`
	  ▶ is set to `SendStaticExternal` for signals sent and
	  ▶ is set to `ReceiveUnqueuedExternal` for signals received.
	▶ `ComSignalSystemSignalShortName`
	  This vendor specific parameter is set to the original name of the signal as defined in the LDF file.
	▶ `ComSignalInitValue`
	  This parameter is set to the value as defined in the LDF file. Byte arrays are not supported.

`ComNetworkSignal`	For every signal sent or received by the master node, the LDF Importer creates a `ComNetworkSignal` container named `NW<PREFIX><name><INSTSUFFIX>`, where `<name>` is the name of the signal in the LDF file.
	Every `ComNetworkSignal` container references the `ComIPdu` container of the PDU that the signal is assigned to and has the following parameters set:
	▶ `ComNetworkSignalSize` is set to the signal bitlength from the LDF file
	▶ `ComNetworkSignalBitPosition` is set to the start bit as defined in the LDF file
	▶ `ComNetworkSignalEndianess`
	  ▶ is set to `Opaque` if signal is a byte array;
	  ▶ is set to `LittleEndian` else
	▶ `ComNetworkSignalDirection`
	  ▶ is set to `Sent` for signals sent and
	  ▶ is set to `Received` for signals received.

	▶ `ComSignalInitValue`: this parameter is set to the value as defined in the LDF file. Byte arrays are not supported.

Autosar 2.1 EcuC module

The LDF Importer creates an EcuC configuration, which contains a global PDU for all unconditional frames that are sent or received by the master node.

`PduCollection/Pdu`	For every unconditional frame sent or received by the master node selected, the LDF Importer creates a `PduCollection/Pdu` container named `<PREFIX><name><INSTSUFFIX>`, where `<name>` is the name of the frame in the LDF file.

Autosar 3.0 Lin module

The LDF Importer handles Autosar 3.0 LIN modules in the same way as Autosar 2.1 LIN modules. Refer to Section: Autosar 2.1 Lin module for details.

Autosar 3.0 LinIf module

The LDF Importer imports all frames and schedule tables found in the LDF file into the LinIf module configuration.

`LinIfGlobalConfig`	One LinIfGlobalConfig container with the name `LinIfGlobalConfig` is created. The importer sets the following parameters: ▶ `LinIfTimeBase`: If the ECU is connected to a single channel, the value is set to the time base of the LIN Master node connecting the ECU to the LIN channel. If the ECU is connected to multiple LIN channels, `LinIfTimeBase` is calculated in two steps: 1. In the first step, the time base parameter of all connecting LIN Master nodes is multiplied by `100000`. This converts the unit from `[s]` to tens of `[us]`; the decimal places are subtracted from the product. 2. In the second step, the greatest common divisor (gcd) of all converted parameter values is calculated, then the gcd is divided by `100000` to obtain it in units of `[s]` again. The importer uses this value for the configuration of `LinIfTimeBase`.

LinIfGlobalConfig/LinIfChannel	One LinIfChannel container with the naming format <PREFIX><Cluster>is created. The importer sets the following parameters: ► LinIfChannelRef: References the corresponding channel in the LIN Driver module.
LinIfChannel/LinIfFrame	For every unconditional, sporadic, and event-triggered frame, a LinFrame container is created with the naming format <PREFIX><name><INSTSUFFIX>. <name> is the name of the frame in the LDF file. The importer sets the following parameters: ► LinIfChecksumType: is set to ENHANCED, if the sender and all receivers are Lin 2.x nodes (i.e. nodes conforming to LIN specification 2.0 or higher), otherwise to CLASSIC. For sporadic and event-triggered frames, the checksum type of one of the substituted frames is set. ► LinIfFrameType: is set to UNCONDITIONAL, SPORADIC or EVENT_TRIGGERED respectively, depending on the frame definition in the LDF file. ► LinIfLength: is set to the length of the frame as defined in the LDF file. For sporadic and event-triggered frames, the frame length of one of the substituted frames is set. ► LinIfPid: is set to the protected identifier of the frame derived by the identifier provided in the LDF file. For sporadic frames, this parameter is not set. ► LinIfSubstitutionFrames: references to the substituted frames are set in this container list using the parameter LinIfSubstitutionFrameRef. ► LinIfFixedFrameSdu: Only set for MasterRequestFrames and SlaveResponseFrames. Describes a complete LIN frame (8 bytes) as part of a configuration. Every byte is represented by a single LinIfFixedFrameSdu container. Parameter LinIfFixedFrameSduBytePos represents the position of the byte, LinIfFixedFrameSduByteVal represents the value of the byte. ► LinIfFrameName: set to frame name as defined in the LDF file. Additionally, the LinIfPduDirection choice container is set to LinIfTxPdu for outgoing frames and to LinIfRxPdu for incoming frames. For unconditional frames a reference to the global PDU is set in the EcuC module. Outgoing frames have this reference set in LinIfTxPduRef, incoming frames in LinIfRxPduRef.
LinIfChannel/LinIfScheduleTable	For every schedule table defined in the LDF file, a LinIfScheduleTable container is created with the naming format <PREFIX><name>. <name> is the name of the schedule table in the LDF file. For every schedule entry, a LinIfEntry container is created. The importer sets the following parameters:

	▶ `LinIfFrameRef`: is set to reference the frame which is transmitted within the entry as defined in the LDF file. ▶ `LinIfDelay`: is set to the duration of the frame slot as defined in the LDF file. ▶ `LinIfEntryIndex`: is set to position of the entry within the schedule table as defined in the LDF file.
`LinIfChannel/LinIf-Master`	For the master node as defined in the LDF file a container is created and its name is set to `LinIfMaster`. The importer sets the following parameters: ▶ `LinIfJitter`: this parameter is defined in units of ms in the LDF file; here it is converted to s. ▶ `LinIfChannelRef`: set to the only `LinIfChannel` created during the import.
`LinIfChannel/LinIf-Slave`	For every Slave node as defined in LDF file a container is created with the naming format `<PREFIX><name><INSTSUFFIX>`. `<name>` is the name of the slave in the LDF file. The importer sets the following parameters: ▶ `LinIfConfiguredNad`: as defined in the LDF file. ▶ `LinIfProtocolVersion`: as defined in the LDF file. ▶ `LinIfSupplierId`: as defined in the product-ID in the LDF file. ▶ `LinIfFunctionId`: as defined in the product-ID in the LDF file. ▶ `LinIfVariant`: as defined in the product-ID in the LDF file. ▶ `LinIfResponseErrorBitPos`: is set to the offset of the `response_error` signal within its transmitting frame as defined in the LDF file. ▶ `LinIfResponseErrorFrameRef`: is set to reference the frame containing the `response_error` signal of the Slave node as defined in the LDF file.

Autosar 3.0 PduR module

The LDF Importer handles Autosar 3.0 PduR modules in the same way as Autosar 2.1 PduR modules. Refer to Section: Autosar 2.1 PduR module for details.

Autosar 3.0 Com module

The Com configuration created by the LDF Importer contains all unconditional frames that are sent or received by the master node as well as all signals that are contained in these frames.

ComIPduGroup PDU groups	If not already present, two general PDU groups are created. They are called RXPDUS_GLOBAL and TXPDUS_GLOBAL. Additionally, two sub-PDU groups are created; they are called RGCluster and TGCluster. These sub-PDU groups are assigned as subgroup to their matching general group.
ComIPdu	For every unconditional frame sent or received by the master node a ComIPdu container is created. It is named PD<PREFIX><name><INSTSUFFIX>. <name> is the name of the frame in the LDF file. Every I-PDU references the associated global PDU (see container PduCollection/Pdu in the ECUC module, Section: Autosar 3.0 EcuC module). The following parameters are set for every I-PDU: **Parameter** / **Value** ComIPduSize — is set to the size of the frame as defined in the LDF file. ComIPduSignalRef — For every Ecu signal, which the current controller receives or sends, one reference to the reference list is added to the corresponding ComSignal container. All I-PDUs that are received by the master node reference the RGCluster ComIPduGroup and have the following parameters set: **Parameter** / **Value** ComIPduDirection — is set to RECEIVE. All I-PDUs that are sent by the master node reference the TGCluster ComIPduGroup and have the following parameters set: **Parameter** / **Value** ComIPduDirection — is set to SEND.
ComSignal	For every signal sent or received by the master node, the LDF Importer creates a ComSignal container named SG<PREFIX><name><INSTSUFFIX>. <name> is the name of the signal in the LDF file. Every ComSignal has the following parameters set: **Parameter** / **Value** ComSignalLength — is set to the number of bytes required for the signal if the parameter ComSignalType is set to UINT8_N. ComSignalType — ► is set to BOOLEAN, if the signal size is one bit;

	Parameter	Value
		▶ is set to `UINT8`for signals with a size of less than or equal to 8 bit; ▶ is set to `UINT16`for signals with a size of more than 8 bit; ▶ is set to `UINT8_N` for byte array signals
	`ComBitSize`	is set to the signal length as defined in the LDF file
	`ComBitPosition`	is set to the start bit as defined in the LDF file
	`ComSignalEndianess`	▶ is always set to `LITTLE_ENDIAN` (in LIN everything is), but ▶ for signals with a `ComSignalType` of `UINT8_N` it is set to `OPAQUE`
	`ComSignalSystemSig-nalShortName`	This vendor-specific parameter is set to the original name of the signal as defined in the LDF file.
	`SystemTemplateSys-temSignalRef`	This reference parameter is set to the corresponding `ISignalToIPduMapping` container of the imported system model.
	`ComSignalInitValue`	is set to the value as defined in the LDF file. Byte arrays are not supported.

Autosar 3.0 EcuC module

The LDF Importer creates an EcuC configuration, which contains a global PDU for all unconditional frames that are sent or received by the master node.

`PduCollection/Pdu`	The LDF Importer creates a `PduCollection/Pdu` container named `<PRE-FIX><name><INSTSUFFIX>`, where `<name>` is the name of the frame in the LDF file. The parameter `PduLength` is set to the length of the frame as defined in the LDF file multiplied by eight.

4.11.2.7. Importing from the Fibex format

4.11.2.7.1. Overview

EB tresos Studio provides an integrated Fibex Importer to import configurations from Fibex files.

The EB tresos Studio Fibex Importer is able to read a Fibex file, fill in the system model of a project and to configure one or more of the following modules:

- `Com` (`Communication`)
- `PduR` (`PDU Router`)
- `FrIf` (`FlexRay Interface`)
- `Fr` (`FlexRay Driver`)
- `EcuC` (`EcuC Module`)
- `Can` (`Can Driver`)
- `CanIf` (`CAN Interface`)

Supported Fibex versions are: 1.1.5a, 1.2.0a, 2.0.0b, 2.0.0d, 2.0.1, 3.0.0, 3.1.0. For detailed background information please refer to the Fibex specification available at http://www.asam.net.

4.11.2.7.2. Background information

Limitations

Known restrictions/limitations:

- Only FlexRay and CAN clusters are imported.
- CAN clusters are only imported from Fibex 3.x.
- SIGNAL-GROUPS (in Fibex <= 2.0.1) are imported as PDUs.
- SIGNAL-GROUPS (in Fibex 3.x) cannot be imported, since these Fibex versions defined them in the wrong context (within frames instead of PDUs).
- No gateway information is imported from Fibex.
- No multiplexers are imported from Fibex.
- Only frames and PDUs of the type APPLICATION are imported.
- The import of PDU transmission modes into the `Com` module is only supported from Fibex 3.x files.

- If a SIGNAL-GROUP (imported as PDU in Fibex <= 2.0.1 files) is assigned to multiple frames with different endianess in a Fibex document, the endianess of the signals contained in the PDU is undefined.

4.11.2.7.3. Importing Fibex configurations

The Fibex Importer enables importing configurations into the system model and into the ECU configuration of the project. The GUI of the importer allows to enter parameters for both these import scenarios.

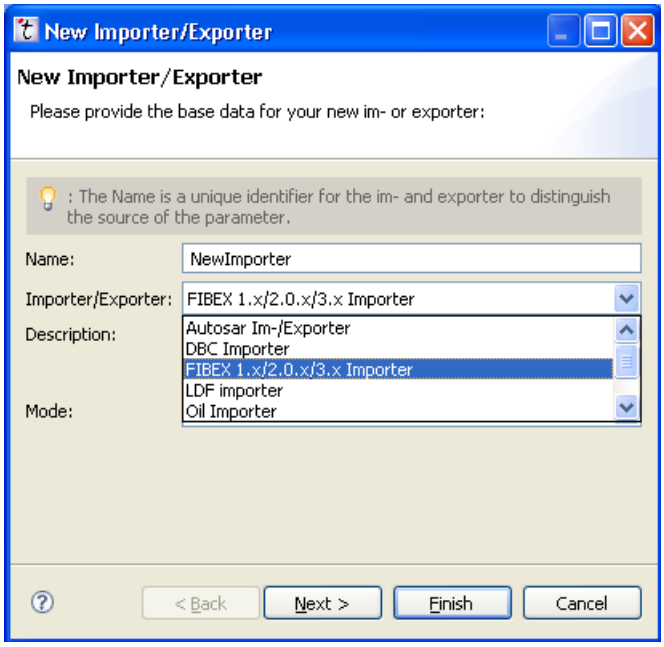
To import configurations from the Fibex format:

- Refer to Section 4.11.2: Importing and exporting configuration data.
- Follow the instructions of Step 1 until you arrive at the window *Create, manage and run im- and exporters*.

In the window *Create, manage and run im- and exporters*:

- Select the **Create a new im- or exporter** button.

The *New Importer/Exporter* window opens up.



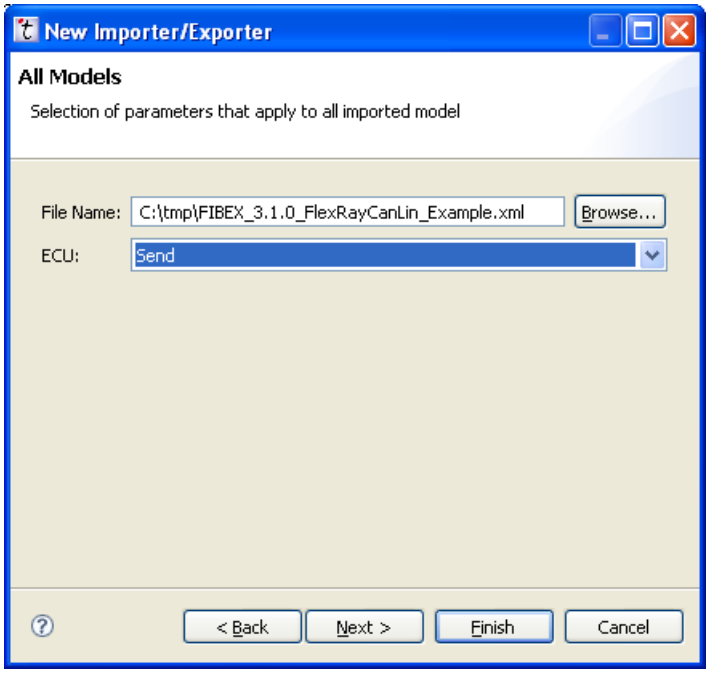
To enter the basic information for the new importer or exporter configuration:

- Select the *Name* textbox.
- Enter a name for your new configuration.
- Select *Fibex 1.x/2.0.x/3.x Importer* from the **Importer/Exporter** dropdown listbox.
- Select **Import only** from the *Mode* dropdown listbox.

The Fibex Importer can only be used to import files. Exporting to Fibex format is not possible.

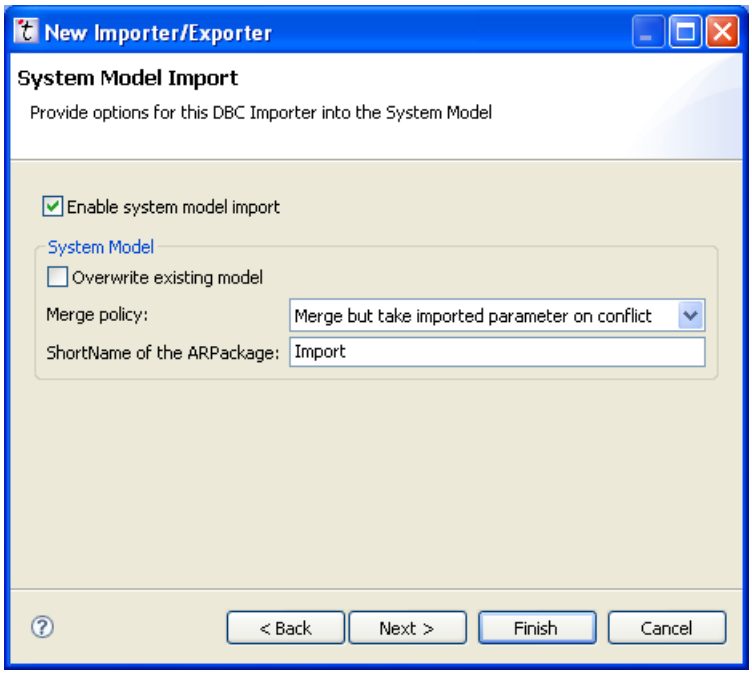
- Select **Next**.

On the following page generic information can be entered.



- Use the **File Name** file selection entry to select the Fibex file that should be imported.
- Via the **ECU** dropdown listbox the ECU that should be imported from the Fibex file must be selected.

Press the **Next** button to switch to a page where parameters for the system model import can be entered.

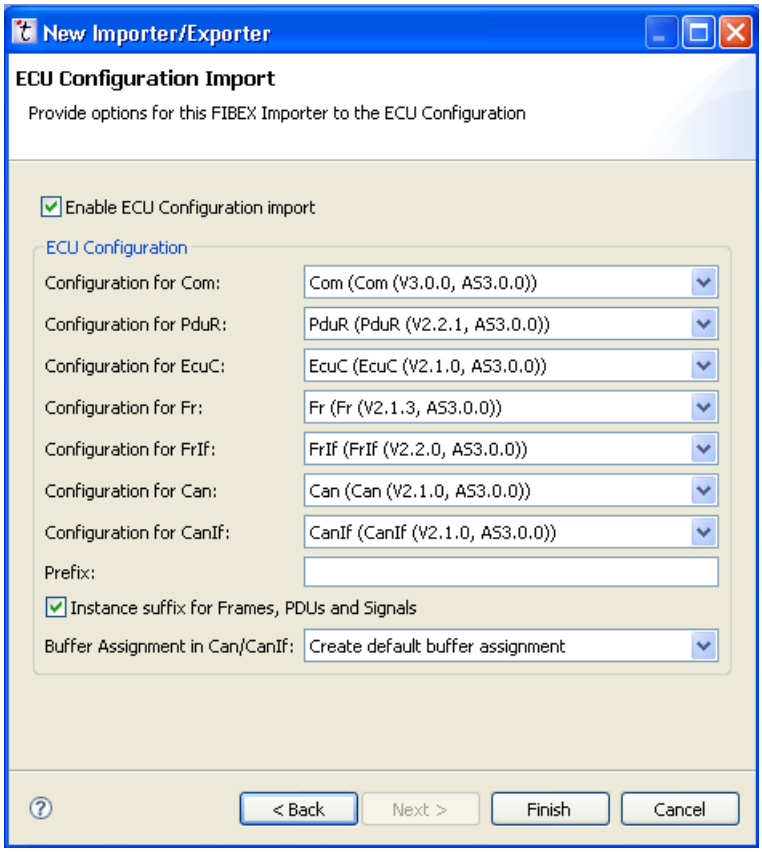


- To enable system model import into the project, check the *Enable system model import* checkbox. If the project has no system model, the complete page including the checkbox is greyed out.
- To completely replace the system model in the project, check the checkbox *Overwrite existing model*.
- To provide a hint for the merge algorithm that merges the imported configuration into the system model of the project, select an entry from the *Merge policy* dropdown listbox .
- To select the Autosar SHORT-NAME of the Autosar AR-PACKAGE into which the system model of the source file is to be imported, enter the Autosar SHORT-NAME into the **ShortName of the ARPackage** textbox.
- Select **Next** to proceed to the next configuration window.

The *ECU Configuration Import* window is provided for entering ECU configuration-specific parameters.

Note

Since the Fibex Importer does not create module configurations, you need to assure that a module configuration is present for every module you would like to import into (`Com`, `PduR`,`FrIf`, `Fr`, `EcuC`, `Can`, `CanIf`).



- The *Enable ECU Configuration Import* checkbox allows to disable ECU configuration import.

- In the *Configuration for Com* dropdown listbox, select the target Com configuration for the import.
- In the *Configuration for PduR* dropdown listbox, select the target PduR configuration for the import.
- In the *Configuration for EcuC* dropdown listbox, select the target EcuC configuration for the import.
- In the *Configuration for Fr* dropdown listbox, select the target Fr configuration for the import.
- In the *Configuration for FrIf* dropdown listbox, select the target FrIf configuration for the import.
- In the *Configuration for Can* dropdown listbox, select the target Can configuration for the import.
- In the *Configuration for CanIf* dropdown listbox, select the target CanIf configuration for the import.
- For details on the *Prefix* textbox see Section 4.11.3.1.2: Name prefixing.
- For details on the *Instance Handling* dropdown listbox see Section 4.11.3.1.3: Instance handling.

To save the importer configuration:

- Select **Finish**.

This saves your importer configuration and closes the *FIBEX Importer Options* window.

- To run the Fibex Importer, select the importer name you just created on the left side of the *Create, manage, and run im- and exporters* window.
- Select **Run Importer**.

The Fibex configuration is being imported.

4.11.2.7.4. Retrieving PDUs from Fibex 2.x and earlier files

Since PDUs are not directly modelled in Fibex until version 3.0, the Fibex Importer uses two different mechanisms for 2.x and earlier Fibex versions to retrieve PDU information out of Fibex instances.

- Signal groups are implicitly imported as PDUs. If a frame contains a signal group, this signal group is imported as PDU. The name of the PDU equals the name of the signal group.
- Frames which do not contain any signal groups receive a default PDU, which occupies the whole payload of the frame. If the frame contains any signals, those signals are assigned to the default PDU. The default PDU obtains the same name as the enclosing frame.

4.11.2.7.5. References

Fibex to Autosar parameter mapping

Overview

The Fibex Importer of EB tresos Studio creates configurations for the Autosar modules `Fr`, `FrIf`, `PduR`, `Com`, `Can`, `CanIf`, and `EcuC`. This section lists the containers and parameters created and on which Fibex parameters and attributes they are based.

Subsections are named by module names or container names respectively. The symbol `<PREFIX>` represents the prefix provided by the user for the import. The symbol `<INSTSUFFIX>` represents the optional, calculated instance suffix for frames, signals, and PDUs.

Background information

Fibex Importer naming convention for container prefixes

The following section lists the name extensions the *Fibex Importer* uses when it creates containers.

- `<name>` represents the name of the signal, PDU or the message respectively as defined in the Fibex file.
- `<clusterName>` represents the name of the FlexRay or CAN communication cluster the ECU selected for import is connected to.
- `<ctrlName>` represents the name of the controller(s) of the ECU which is selected for import.
- `<ftName>` represents the name of a FlexRay frame triggering which triggers the transmission of a frame in the FlexRay network.
- Names may be mangled, for details see Section 4.11.3.1: Naming rules.

Autosar container	Name format
`CanController`	`<PREFIX><ctrlName>`
`CanIfControllerConfig`	`<PREFIX><ctrlName>`
`CanIfDriverConfig`	`<PREFIX>CanIfDriverConfig`
`CanIfNetworkConfig`	`<PREFIX>CanNetwork`
`CanIfRxPduConfig`	`<PREFIX><name><INSTSUFFIX>`

Autosar container	Name format
CanIfTxPduConfig	<PREFIX><name><INSTSUFFIX>
FrController	<PREFIX><ctrlName>
FrIfFrameStructure	f<PREFIX><name><INSTSUFFIX>
FrIfPdu	p<PREFIX><name><INSTSUFFIX>
FrIfCluster	<PREFIX><clusterName>
FrIfController	<PREFIX><ctrlName>
FrIfFrameTriggering	t<PREFIX><ftName><INSTSUFFIX>
PduRRoutingPath	<PREFIX><name><INSTSUFFIX>
ComIPdu	<PREFIX><name><INSTSUFFIX>
ComIPduGroup	▶ ImportedRxPdus for Autosar 2.1 configurations, additionally cluster specific Rx subgroups for Autosar 3.0 configurations ▶ ImportedTxPdus for Autosar 2.1 configurations, additionally cluster specific Tx subgroups for Autosar 3.0 configurations
ComNetworkSignal	NW<PREFIX><name><INSTSUFFIX>
ComSignal	SG<PREFIX><name><INSTSUFFIX>
Pdu (EcuC module)	<PREFIX><name><INSTSUFFIX>

The following sections outline how the Fibex Importer performs the parameter mapping for Autosar 2.1 and Autosar 3.0 modules.

Autosar 2.1 Can module

The Fibex Importer creates a basic CAN Driver configuration without hardware-specific details.

CanController	A CanConfigSet/CanController container is created with the name <PREFIX><name>, where <name> is the name of the controller selected. The following parameters are set: CanControllerBaudRate, CanControllerSeg1, and CanControllerSeg2. For this controller two filter masks are created: AcceptAllStd and AcceptAllExt. These are set to the filter values necessary to accept all standard and extended identifiers, which can either be: ▶ 0x7ff and 0x1fffffff if the CAN Driver implementation assumes a set bit as don't care bit value, or

	► `0x0` for both standard and extended identifiers if the don't care bit value is a cleared bit.
CanHardwareObject	If the *Buffer Assignment in Can/CanIf* parameter has been set to *Create default buffer assignment*, the Fibex Importer creates up to four hardware objects in the container `CanConfigSet`: 1. a transmit object (*Hardware Transmit Handle* , *HTH*) for messages with standard identifiers, 2. a transmit object for messages with extended identifiers, 3. a receive object (*Hardware Receive Handle* , *HRH*) for messages with standard identifiers, and 4. a receive object for messages with extended identifiers. The names of these hardware objects are: 1. `TxStd<PREFIX><Controllername>` (HthStandard) 2. `TxExt<PREFIX><Controllername>` (HthExtended) 3. `RxStd<PREFIX><Controllername>` (HrhStandard) 4. `RxExt<PREFIX><Controllername>` (HrhExtended) Each of them is created only if there is at least one import message that requires it. The Fibex Importer automatically links these HRHs and HTHs to the controller created (see `CanIfControllerConfig`). The `CanObjectId` parameter is set in such a way that the parameter values are zero-based and dense and HrhStandard and HrhExtended objects obtain lower values than HthStandard and HthExtended objects.

Autosar 2.1 CanIf module

The CAN Interface configuration imported by the Fibex Importer contains the definition of all messages that are sent or received by the controller selected. Additionally it includes the `CanIfDriverConfig`, `CanIfControllerConfig`, and `CanIfNetworkConfig` containers.

CanIfDriverConfig	If a `CanIfDriverConfig` does not yet exist, the container is created with the name `<PREFIX>CanIfDriverConfig`. If a subcontainer already exists, the content of this container is configured without changing its name. A subcontainer can be e.g. the default container `CanIfDriverConfig_0`. If the *Buffer Assignment in Can/CanIf* parameter has been set to *Create default buffer assignment*, up to four hardware objects are created. The exact number depends on the actual configuration (similar to what is imported into `CanHardwareO-`

	bject in the CAN module, Section: Autosar 2.1 Can module). The hardware objects are configured for sending and receiving CAN messages with standard and extended identifiers. They reference the respective hardware object in the CAN Driver configuration and the CanIfControllerConfig created. The parameters CanIfHthType and CanIfHrhType are set to BasicCan.
CanIfControllerConfig	A CanIfControllerConfig container is created with the name <PREFIX><name>, where <name> is the name of the controller selected. The controller created references ▶ the CAN Driver controller created (see CanController in the Can module, Section: Autosar 2.1 Can module), ▶ the CAN Interface driver configuration created (see container CanIfDriverConfig in the CanIf module, Section: Autosar 2.1 CanIf module), ▶ and the CAN Interface network configuration created (see container CanIfNetworkConfig in the CanIf module, Section: Autosar 2.1 CanIf module).
CanIfNetworkConfig	The CanIfNetworkConfig container is created with the name <PREFIX>CanNetwork.
CanIfInitConfigSet	The container CanIfInitConfigSet is created. This container contains the following subcontainers. ▶ CanIfInitNetworkConfig ▶ CanIfInitControllerConfig ▶ CanIfInitDriverConfig ▶ CanIfInitConfig Below the CanIfInitNetworkConfig list container, one subcontainer is created with the name <PREFIX>CanNetworkInitNW. ▶ The parameter CanIfControllerConfigSet is set to CANIF_NETWORK_0_CONFIG_0. ▶ The link parameter CanIfNetworkRef is set to the enclosing container. ▶ The container CanIfInitControllerConfig is left untouched by the Fibex Importer. ▶ If a container below CanIfInitDriverConfig already exists, its CanIfDriverRef reference is set to the CanIfDriverConfig container. If it does not exist yet, a new container named <PREFIX>CanIfInitDriverConfig is created and CanIfDriverRef is set accordingly.

	▶ If a container below `CanIfInitConfig` already exists, this container is used for inserting the Rx and Tx PDUs. ▶ If such a container does not exist, a new container with the name `CanIfInit-Config` is created. The following sections describe how the Fibex Importer fills this container.
CanIfInitConfig	If a `CanIfInitConfig` entry in the `CanIfInitConfig` list does not yet exist, the container is created with the name `CanIfInitConfig`. If a subcontainer already exists, the content of this container is configured without changing its name.

CanIfRxPduConfig	For every PDU received by the controller, a `CanIfRxPduConfig` container is created below `CanIfInitConfigSet/CanIfInitConfig`. This container is named `<PREFIX><name><INSTSUFFIX>`, where `<name>` is the name of the PDU in the Fibex file.
	
	The following parameters are set:
	
	`CanIfRxPduRxIndiUser`
	is set to `PduR`
	
	`CanIfRxPduRxIndiCallOutFun`
	is set to `PduR_CanIfRxIndication`
	
	`CanIfRxPduIdCanIdType`
	is set to `Standard` or `Extended` as defined in the Fibex file
	
	`CanIfCanRxPduCanId`
	is set to the CAN identifier as defined in the Fibex file
	
	`CanIfCanRxPduDlc`
	is set to the PDU's byte length as defined in the Fibex file
	
	Additionally the following references are set:
	
	▶ If the *Buffer Assignment in Can/CanIf* parameter has been set to *Create default buffer assignment*, the matching hardware object of the `CanIfDriverConfig` created is referenced (see container `CanIfDriverConfig` in the CanIf module, Section: Autosar 2.1 CanIf module).
	▶ The `CanIfNetworkConfig` created is referenced (see container `CanIfNet-workConfig` in the CanIf module, Section: Autosar 2.1 CanIf module),
	▶ as well as the global PDU container in the EcuC (see Section: Autosar 2.1 EcuC module).

CanIfTxPduConfig	For every PDU sent by the controller, a `CanIfTxPduConfig` container is created below `CanIfInitConfigSet/CanIfInitConfig`. This container is named

	`<PREFIX><name><INSTSUFFIX>`, where `<name>` is the name of the PDU in the Fibex file.
	The following parameters are set:
	`CanIfTxUserType`
	is set to `PduR`
	`CanIfTxPduTxConfirmCallOut`
	is set to `PduR_CanIfTxConfirmation`
	`CanIfTxPduIdCanIdType`
	is set to `Standard` or `Extended` as defined in the Fibex file
	`CanIfCanTxPduCanId`
	is set to the CAN identifier as defined in the Fibex file
	`CanIfCanTxPduDlc`
	is set to the PDU's byte length as defined in the Fibex file
	► If the *Buffer Assignment in Can/CanIf* parameter has been set to *Create default buffer assignment*, the matching hardware object of the `CanIfDriverConfig` created is referenced (see container `CanIfDriverConfig` in the CanIf module, Section: Autosar 2.1 CanIf module).
	► The `CanIfNetworkConfig` created is referenced (see container `CanIfNetworkConfig` in the CanIf module, Section: Autosar 2.1 CanIf module),
	► as well as the global PDU associated with this message (see Section: Autosar 2.1 EcuC module).

Autosar 2.1 Fr module

A FrController container is created with the name `<PREFIX><name>`, where `<name>` is the name of the FlexRay controller.

If it does not already exist, one `FrAbsoluteTimer` subcontainer is created for every FrController.

With a few exceptions, all local FlexRay parameters (i.e. FlexRay parameters which may be different among the FlexRay controllers attached to one FlexRay network) are imported without any changes apart from different units or enum values. The exceptions are listed below

The following parameters are not imported:

Autosar parameter	Reason
FrCtrlClock	This parameter is not part of the Fibex FlexRay controller parameter set, and can therefore not be imported. Moreover, this parameter is not necessary for Fr configuration an may therefore be left at the default value.
FrCtrlIdx	This parameter is not part of the Fibex FlexRay controller parameter set, and can therefore not be imported. The parameter is calculated in a dedicated configuration step after the import took place. For detailed information, refer to Section 4.9.3.4: Configuring the *Calculate Handle-Ids* wizard.
FrIsLeadingColdstarter	This parameter is not part of the Fibex FlexRay controller parameter set, and must be added by hand.

Autosar 2.1 FrIf module

<table>
<tr><td>FrIfCluster</td><td>A FrIfCluster container is created with the name <PREFIX><name>, where <name> is the name of the FlexRay cluster.

With a few exceptions, all global FlexRay parameters (i.e. FlexRay parameters are the same for all FlexRay controllers attached to one FlexRay network) are imported without any changes apart from different units or enum values. The exceptions are listed below

The following parameters are not imported:
<table>
<tr><th>Autosar parameter</th><th>Reason</th></tr>
<tr><td>FrIfClstIdx</td><td>This parameter is not part of the Fibex FlexRay cluster parameter set, and can therefore not be imported. It is calculated in a dedicated configuration step after the import took place. For detailed information, refer to Section 4.9.3.4: Configuring the Calculate Handle-Ids wizard.</td></tr>
<tr><td>FrIfMainFunctionPeriod</td><td>Since this parameter is not part of the Fibex parameter set, it must be configured manually.</td></tr>
<tr><td>FrIfMaxIsrDelay</td><td>Since this parameter is not part of the Fibex parameter set, it must be configured manually.</td></tr>
<tr><td>GClusterDriftDamping,
GOffsetCorrectionMax</td><td>These parameters define constraints on other FlexRay parameters, however they are not directly used to configure a FlexRay communication controller. Therefore these parameter values are left empty.</td></tr>
</table></td></tr>
<tr><td>FrIfCluster/FrIfController</td><td>One FrIfCluster/FrIfController container is created for every FlexRay communication controller which is assigned to the selected ECU and attached to the FlexRay cluster.</td></tr>
</table>

	The container is created with the name `<PREFIX><name>`, where `<name>` is the name of the FlexRay communication controller. Below every created FrIfCluster/FrIfController container, one FrIfAbsTimer subcontainer is added, which references its `Fr` counterpart in the `FrIfFrAbsTimerRef` parameter. The following parameters are imported: **Autosar parameter** / **Value** FrIfFrCtrlRef / References the corresponding container in the Fr module, see Section: Autosar 2.1 Fr module For every frame triggering which is sent or received by the communication controller, one FrIfFrameTriggering subcontainer is created.
FrIfCluster/FrIfController/FrIfFrameTriggering	The following FrIfFrameTriggering parameters are imported: **Autosar parameter** / **Reason** FrIfBaseCycle / Is set to the base cycle parameter of the Frame Triggering in the Fibex file. FrIfCycleRepetition / Is set to the cycle repetition parameter of the Frame Triggering in the Fibex file. FrIfChannel / Is set to the FlexRay channel the Frame Triggering is attached to in the Fibex file. FrIfSlotId / Is set to the FlexRay slot of the Frame Triggering. FrIfFrameStructureRef / References the FrIfFrameStructure container representing the container transmitted via this Frame Triggering.
FrIfPdu	Only PDUs sent/received by the selected ECU are imported. The name of the container is `p<PREFIX><name><INSTSUFFIX>`, where <name> is the name of the PDU, see Section 4.11.2.7.4: Retrieving PDUs from Fibex 2.x and earlier files. The following FrIfPdu parameters are imported: **Autosar parameter** / **Value** FrIfPduLength / The parameter FrIfPduLength is determined by the PDUs byte length, see Section 4.11.2.7.4: Retrieving PDUs from Fibex 2.x and earlier files. FrIfPduRef / References the corresponding global PDU container in the EcuC (see Section: Autosar 2.1 EcuC module). FrIfPduDirection / Can be either FrIfTxPdu or FrIfRxPdu.

FrIfFrameStructure	Fibex frames which are sent or received by the selected ECU are imported as FrIf-FrameStructure. The name of the container is `f<PREFIX><name><INSTSUFFIX>`, where <name> is the name of the frame in the Fibex document. The following FrIfFrameStructure parameters are imported: **Autosar parameter** / **Value** FrIfLSduLength / Contains the length of the frame in bytes. For every sent or received PDU contained in the frame, one FrIfPdusInFrame sub-container is created.
FrIfPdusInFrame	This entry represents one frame to PDU mapping. The name of the container is `<PREFIX><name><INSTSUFFIX>`, where <name> is the name of the PDU, see Section 4.11.2.7.4: Retrieving PDUs from Fibex 2.x and earlier files. The following FrIfPdusInFrame parameters are imported: **Autosar parameter** / **Value** FrIfPduOffset / Contains the offset of the PDU within its enclosing frame in bytes. FrIfFrIfPduRef / Contains a reference to the corresponding FrIfPdu entry. FrIfPduUpdateBitOffset / If the PDU has assigned an update bit in the enclosing frame, the offset of the update bit is configured here. Depending on various Fibex parameters that have an influence on how the Fibex offset is interpreted, the startbit is converted into Autosar representation, see Section 4.11.3.1.4: Signal offsets. This parameter is only imported from Fibex 3.x files.

Autosar 2.1 PduR module

The PDU Router configuration created by the Fibex Importer contains routing paths for all messages sent or received by the controller selected. All received messages are routed from the FrIf module to the Com module, and all sent messages are routed from the Com module to the FrIf module.

PduRGlobalConfig/PduRRoutingPath	The Fibex Importer creates a `PduRGlobalConfig/PduRRoutingPath` container for every PDU sent or received by the chosen ECU. This container is named after the PDUs name in the Fibex file in accordance with the name mangling described in Section 4.11.3.1: Naming rules.

	The importer creates a PduRSrcPdu and a PduRDestPdu container for this PduRRoutingPath. Both the source and the destination PDU containers are set up to reference the global PDU associated with the frame (see PduCollection/Pdu in Section: Autosar 2.1 EcuC module). The Fibex Importer sets the parameter SduLength to the bytelength of the SDU containing the PDU.

Autosar 2.1 Com module

The Fibex Importer COM configuration contains all messages sent or received by the controller. It also contains all signals contained within these messages.

ComIPduGroup PDU groups	If not already present, two general PDU groups are created. They are called RXPDUS_GLOBAL and TXPDUS_GLOBAL. Two sub PDU groups called RG<PREFIX><name> and TG<PREFIX><name> are additionally created and assigned as subgroup to their matching general group. <name> represents the name of the imported FlexRay network as defined in the Fibex file.
ComIPdu	For every PDU sent or received by the ECU selected a ComIPdu container is created. The container is named PD<PREFIX><name><INSTSUFFIX> , with <name> being the name of the PDU in the Fibex file. Every I-PDU references its associated global PDU (see PduCollection/Pdu in Section: Autosar 2.1 EcuC module) and has the following parameters set: **Parameter** / **Value** ComIPduSize / set to the PDU length as defined in the Fibex file. All I-PDUs that are received by the controller selected reference the associated ComIPduGroup RG<PREFIX><name> (<name> being the name of the imported FlexRay network as defined in the Fibex file) and have the following parameters set: **Parameter** / **Value** ComIPduDirection / is set to Received All I-PDUs that are sent by the controller selected reference the associated ComIPduGroup TG<PREFIX><name> (<name> being the name of the imported FlexRay network as defined in the Fibex file) and have the following parameters set: **Parameter** / **Value** ComIPduDirection / is set to Sent

<table>
<tr><td></td><td>The following parameter is only configured for sent I-PDUs and when you import from Fibex 3.x.
<table>
<tr><th>Parameter</th><th>Value</th></tr>
<tr><td>ComIPduMinimumDelay-Factor</td><td>Calculated from the DEBOUNCE-TIME-RANGE of the first EVENT-CONTROLLED-TIMING of the PDU's PDU-TRIGGERING. See the interval calculation formula outlined in Section: Extensions to the Autosar 2.1 Com module for details.</td></tr>
</table></td></tr>
<tr><td>ComIPdu/ComTransmissionModeTrue</td><td>The following parameters are only configured for sent I-PDUs and when you import from Fibex 3.x.

ComTxModeMode is set to one of the following values for outgoing ComIPdus :

▶ Direct, if only one EVENT-CONTROLLED-TIMING is defined in the PDU's PDU-TRIGGERING
▶ Periodic, if only one CYCLIC-TIMING is defined in the PDU's PDU-TRIGGERING
▶ Mixed, if both EVENT-CONTROLLED-TIMING and CYCLIC-TIMING are defined in the PDU's PDU-TRIGGERING

If ComTxModeMode is set to Direct, the MIN parameter ComTxModeRepetition-PeriodFactor is calculated from the DEBOUNCE-TIME-RANGE value according to the interval calculation formula outlined in Section: Extensions to the Autosar 2.1 Com module. ComTxModeNumberOfRepetitions is set to FINAL-REPETITIONS.

If ComTxModeMode is set to Periodic, the MAX parameter ComTxModeTimePeri-odFactor is calculated from the REPEATING-TIME-RANGE value according to the interval calculation formula outlined in Section: Extensions to the Autosar 2.1 Com module. The MIN parameter ComTxModeTimeOffsetFactor is derived from the STARTING-TIME-RANGE value. ComTxModeNumberOfRepetitions is set to FINAL-REPETITIONS.

If ComTxModeMode is set to Mixed, all parameters for Direct and Cyclic mode are set.</td></tr>
<tr><td>ComSignal</td><td>For every signal sent or received by the ECU selected, a ComSignal container is created. The container is named SG<PREFIX><name><INSTSUFFIX>, whith <name> being the name of the signal in the Fibex file.

Every ComSignal container references the associated ComNetworkSignal container and has the following parameters set:</td></tr>
</table>

	Parameter	**Value**
	ComSignalLength	is set to the number of bytes required for the signal if the parameter ComSignalType is set to UInt8_n.
	ComSignalType	▶ is set to UInt8/SInt8for signals with Fibex datatype A_UINT8/A_INT8; ▶ is set to UInt16/SInt16for signals with Fibex datatype A_UINT16/A_INT16; ▶ is set to UInt32/SInt32for signals with Fibex datatype A_UINT32/A_INT32; ▶ is set to UInt8_n for signals with a different Fibex datatype
	ComSignalProperty	▶ is set to SendStaticExternal for signals sent ▶ ReceiveUnqueuedExternal for signals received
	ComSignalSystemSig-nalShortName	This vendor specific parameter is set to the original name of the signal as defined in the Fibex file.
ComNetworkSignal	For every signal sent or received by the controller a ComNetworkSignal container is created. The container is named NW<PREFIX><name><INSTSUFFIX>, with <name> being the name of the signal in the Fibex file. Every ComNetworkSignal container references the ComIPdu container of the PDU the signal is assigned to and has the following parameters set:	
	Parameter	**Value**
	ComNetworkSignalS-ize	is set to the signal length from the Fibex file.
	ComNetworkSignalBit-Position	is set to the start bit as defined in the Fibex file. Depending on various Fibex parameters that have an influence on how the Fibex offset is interpreted, the startbit is converted into Autosar representation, see Section 4.11.3.1.4: Signal offsets.
	ComNetworkSignalEn-dianess	▶ is set to LittleEndian if the enclosing frame is a little-endian frame or to BigEndian if the enclosing frame is a big-endian frame. ▶ for signals that have a ComSignalType of UInt8_n it is set to Opaque.

	Parameter	Value
	ComNetworkSignalDirection	▶ is set to Sent for signals sent ▶ is set to Received for signals received
	ComUpdate/ComUpdateBitPosition	is set to the bit offset of the update bit of the signal within the enclosing PDU. Depending on various Fibex parameters that have an influence on how the Fibex offset is interpreted, the startbit is converted into Autosar representation, see Section 4.11.3.1.4: Signal offsets. This parameter is only imported from Fibex 3.x files.

Autosar 2.1 EcuC module

The EcuC configuration created by the Fibex Importer contains a global PDU for all signal groups/enclosing frames that are sent or received by the selected controller.

PduCollection/Pdu	The Fibex Importer creates a PduCollection/Pdu container named <PREFIX><name><INSTSUFFIX>, where <name> is the name of the PDU in the Fibex file.

Autosar 3.0 Can module

The Fibex Importer handles Autosar 3.0 Can modules in the same way as Autosar 2.1 Can modules with the exception that an additional CanConfigSet container is inserted into the Can module configuration. Into this container, the Can configuration is imported. Refer to Section: Autosar 2.1 Can module for details.

Autosar 3.0 CanIf module

The CAN Interface configuration imported by the Fibex Importer contains the definition of all messages that are sent or received by the controller selected. Additionally it includes the CanIfDriverConfig and CanIfControllerConfig containers.

CanIfDriverConfig	If a CanIfDriverConfig does not yet exist, the container is created with the name <PREFIX>CanIfDriverConfig. If a subcontainer already exists, the content of this container is configured without changing its name. A subcontainer is e.g. the default container CanIfDriverConfig_0). In this container the following are referenced:

	▶ the associated `CanIfInitHohConfig` container and ▶ the `CanGeneral` container of the corresponding `Can` configuration. See Section: Autosar 3.0 Can module.
CanIfControllerConfig	A `CanIfControllerConfig` container is created with the name `<PREFIX><name>`, where `<name>` is the name of the controller selected. The controller created references: ▶ the `CAN Driver` controller created (see `CanController` in the `Can` module, Section: Autosar 3.0 Can module), ▶ the `CAN Interface` driver configuration created (see container `CanIfDriverConfig` in the `CanIf` module, Section: Autosar 3.0 CanIf module), ▶ and the corresponding `CanIfInitControllerConfig` container, see Section: Autosar 3.0 CanIf module).
CanIfInitConfiguration	If a `CanIfInitConfiguration` container does not yet exist, it is created with the name `CanIfInitConfiguration`. If the container already exists (e.g. the default container `CanIf_InitConfig_0`, the content of this container is configured without changing its name. The following subcontainers are created: ▶ CanIfInitHohConfig ▶ CanIfInitRxPduConfig ▶ CanIfInitTxPduConfig ▶ CanIfInitControllerConfig The following sections describe how the Fibex Importer fills these containers.
CanIfInitHohConfig	If the *Buffer Assignment in Can/CanIf* parameter has been set to *Create default buffer assignment*, up to four hardware objects are created. The exact number depends on the actual configuration (similar to what is imported into `CanHardwareObject` in the CAN module, Section: Autosar 3.0 Can module). The hardware objects are configured for sending and receiving CAN messages with standard and extended identifiers. They reference the respective hardware object in the `CAN Driver` configuration and the `CanIfControllerConfig` created. The parameters `CanIfHthType` and `CanIfHrhType` are set to `BASIC_CAN`.
CanIfRxPduConfig	For every message received by the controller, a `CanIfRxPduConfig` container is created. This container is named `<PREFIX><name><INSTSUFFIX>`, where `<name>` is the name of the PDU in the Fibex file. The following parameters are set:

	`CanIfRxUserType`
	is set to `PDUR`
	
	`CanIfUserRxIndication`
	is set to `PduR_CanIfRxIndication`
	
	`CanIfRxPduIdCanIdType`
	is set to `STANDARD_CAN` or `EXTENDED_CAN` as defined in the Fibex file
	
	`CanIfCanRxPduCanId`
	is set to the CAN identifier as defined in the Fibex file
	
	`CanIfCanRxPduDlc`
	is set to the PDU's byte length as defined in the Fibex file
	
	Additionally the following references are set:
	
	► If the *Buffer Assignment in Can/CanIf* parameter has been set to *Create default buffer assignment*, the matching hardware object is referenced (see container `CanIfInitHohConfig` in the CanIf module, Section: Autosar 3.0 CanIf module).
	► A link to the global PDU container in the `EcuC` (see Section: Autosar 3.0 EcuC module) is also set.

CanIfTxPduConfig	For every message sent by the controller, a `CanIfTxPduConfig` container is created. This container is named `<PREFIX><name><INSTSUFFIX>`, where `<name>` is the name of the PDU in the Fibex file.
	
	The following parameters are set:
	
	`CanIfTxUserType`
	is set to `PDUR`
	
	`CanIfUserTxConfirmation`
	is set to `PduR_CanIfTxConfirmation`
	
	`CanIfTxPduIdCanIdType`
	is set to `STANDARD_CAN` or `EXTENDED_CAN` as defined in the Fibex file
	
	`CanIfCanTxPduIdCanId`
	is set to the CAN identifier as defined in the Fibex file
	
	`CanIfCanTxPduDlc`
	is set to the PDU's byte length as defined in the Fibex file
	
	Additionally the following references are set:
	
	► If the *Buffer Assignment in Can/CanIf* parameter has been set to *Create default buffer assignment*, the matching hardware object is referenced (see container `CanIfInitHohConfig` in the CanIf module, Section: Autosar 3.0 CanIf module).

	▶ A link to the global PDU container in the EcuC (see Section: Autosar 3.0 EcuC module) is also set.

Autosar 3.0 Fr module

Apart from a slightly different container structure and different values for enumeration parameters, the Fibex Importer imports Autosar 3.0 Fr configurations in the same manner as Autosar 2.1 Fr configurations. Please refer to Section: Autosar 2.1 Fr module for detailed information.

Autosar 3.0 FrIf module

Apart from a slightly different container structure and different values for enumeration parameters, the Fibex Importer imports Autosar 3.0 Fr configurations in the same manner as Autosar 2.1 Fr configurations. Please refer to Section: Autosar 2.1 FrIf module for detailed information.

The Autosar 3.0 FrIf module defines a set of additional `FrIfCluster` parameters, which are not imported because they are not part of the Flexray cluster parameter set of Fibex. Since those parameters are not necessary for the FrIf configuration, the user may leave them unchanged.

The following list outlines those parameters:

- `GAssumedPreciscion`
- `GClusterDriftDamping`
- `GdBitMax`
- `GdBitMin`
- `GdMaxMicrotick`
- `GdMaxPropagationDelay`
- `GdMinPropagationDelay`
- `GOffsetCorrectionMax`

Autosar 3.0 PduR module

The Fibex Importer handles Autosar 3.0 PduR modules in essentially the same way as Autosar 2.1 PduR modules. Refer to Section: Autosar 2.1 PduR module for details.

Autosar 3.0 Com module

The Fibex Importer COM configuration contains all messages sent or received by the controller. It also contains all signals contained within these messages.

<table>
<tr><td>ComIPduGroup PDU groups</td><td>If not already present, two general PDU groups (if not already existing) are created. They are called RXPDUS_GLOBAL and TXPDUS_GLOBAL. Two sub PDU groups called RG<PREFIX><clustername> and TG<PREFIX><clustername> are additionally created and assigned as subgroup to their matching. general group.</td></tr>
<tr><td>ComIPdu</td><td>For every PDU sent or received by the ECU selected a ComIPdu container is created. The container is named PD<PREFIX><name><INSTSUFFIX> , with <name> being the name of the PDU in the Fibex file.

Every I-PDU references its associated global PDU (see PduCollection/Pdu in Section: Autosar 3.0 EcuC module) and has the following parameters set:
<table>
<tr><th>Parameter</th><th>Value</th></tr>
<tr><td>ComIPduSize</td><td>set to the PDUs byte length as defined in the Fibex file.</td></tr>
<tr><td>ComIPduSignalRef</td><td>For every signal that is contained in the Ecu which the current controller actually receives or sends, one reference to the corresponding ComSignal container is added to this reference list.</td></tr>
</table>
All I-PDUs that are received by the ECU selected reference the associated ComIPduGroup RG<PREFIX><clustername> and have the following parameters set:
<table>
<tr><th>Parameter</th><th>Value</th></tr>
<tr><td>ComIPduDirection</td><td>is set to RECEIVE</td></tr>
</table>
All I-PDUs that are sent by the controller selected reference the associated ComIPduGroup TG<PREFIX><clustername> and have the following parameters set:
<table>
<tr><th>Parameter</th><th>Value</th></tr>
<tr><td>ComIPduDirection</td><td>is set to SEND</td></tr>
</table></td></tr>
<tr><td>ComConfig/ComIPdu/ComTxIPdu</td><td>The following parameters are only configured if you import from Fibex 3.x.

The MIN parameter ComTxIPduMinimumDelayTimeFactor is calculated from the DEBOUNCE-TIME-RANGE of the first EVENT-CONTROLLED-TIMING of the PDU's PDU-TRIGGERING. See the interval calculation formula outlined in Section: Extensions to the Autosar 2.1 Com module for details.</td></tr>
<tr><td>ComIPdu/ComTxIPdu/ComTxModeTrue</td><td>The following parameters are only configured if you import from Fibex 3.x.

ComTxModeMode is set to one of the following values for outgoing ComIPdus:</td></tr>
</table>

	1. DIRECT, if only one EVENT-CONTROLLED-TIMING is defined in the PDU's PDU-TRIGGERING 2. PERIODIC, if only one CYCLIC-TIMING is defined in the PDU's PDU-TRIGGERING 3. MIXED, if both EVENT-CONTROLLED-TIMING and CYCLIC-TIMING are defined in the PDU's PDU-TRIGGERING 1. If ComTxModeMode is set to DIRECT, the MIN parameter ComTxModeRepetitionPeriodFactor is calculated from the DEBOUNCE-TIME-RANGE value according to the interval calculation formula outlined in Section: Extensions to the Autosar 2.1 Com module. ComTxModeNumberOfRepetitions is set to FINAL-REPETITIONS. 2. If ComTxModeMode is set to PERIODIC, the MAX parameter ComTxModeTimePeriodFactor is calculated from the REPEATING-TIME-RANGE value according to the interval calculation formula outlined in Section: Extensions to the Autosar 2.1 Com module. The MIN parameter ComTxModeTimeOffsetFactor is derived from the STARTING-TIME-RANGE value. ComTxModeNumberOfRepetitions is set to FINAL-REPETITIONS. 3. If ComTxModeMode is set to MIXED, all parameters for DIRECT and CYCLIC mode are set.
ComSignal	For every signal sent or received by the controller selected, a ComSignal container is created. The container is named SG<PREFIX><name><INSTSUFFIX>, whith <name> being the name of the signal in the Fibex file. Every ComSignal has the following parameters set: **Parameter** / **Value** ComSignalLength — is set to the number of bytes required for the signal if the parameter ComSignalType is set to UINT8_N. ComSignalType — ▶ is set to UINT8/SINT8for signals with Fibex datatype A_UINT8/A_INT8; ▶ is set to UINT16/SINT16for signals with Fibex datatype A_UINT16/A_INT16; ▶ is set to UINT32/SINT32for signals with Fibex datatype A_UINT32/A_INT32; ▶ is set to UINT8_N for signals with a different Fibex datatype

	Parameter	Value
	ComBitSize	is set to the signal bit length as defined in the Fibex file.
	ComBitPosition	is set to the start bit as defined in the Fibex file. Depending on various Fibex parameters that have an influence on how the Fibex offset is interpreted, the startbit is converted into Autosar representation, see Section 4.11.3.1.4: Signal offsets.
	ComSignalEndianess	▶ is set to LITTLE_ENDIAN if the enclosing frame is a little-endian frame or to BIG_ENDIAN if the enclosing frame is a big-endian frame. ▶ for signals with a ComSignalType of UINT8_N it is set to OPAQUE
	ComUpdateBitPosition	This parameter is set to the bit offset of the update bit of the signal within the enclosing PDU. Depending on various Fibex parameters that have an influence on how the Fibex offset is interpreted, the startbit is converted into Autosar representation, see Section 4.11.3.1.4: Signal offsets. This parameter is only imported from Fibex 3.x files.
	ComSignalSystemSignalShortName	This vendor specific parameter is set to the original name of the signal as defined in the Fibex file.
	SystemTemplateSystemSignalRef	This reference parameter is set to the corresponding ISignalToIPduMapping container of the imported system model.

Autosar 3.0 EcuC module

The EcuC configuration created by the Fibex Importer contains a global PDU for all messages that are sent or received by the selected controller.

PduCollection/Pdu	The Fibex Importer creates a PduCollection/Pdu container named <PREFIX><name><INSTSUFFIX>, where <name> is the name of the PDU in the Fibex file. The parameter PduLength is set to the length of the PDU as defined in the Fibex file in units of bits.

4.11.2.8. Importing configurations from the TDB format

4.11.2.8.1. Overview

EB tresos Studio provides an integrated TDB Importer to import a system model as well as ECU configurations from TDB files into the project.

4.11.2.8.2. Background information

When importing into the ECU configuration the EB tresos Studio TDB Importer is able to read a TDB file and to configure one or more of the following modules:

- `Com` (`Communication`)
- `PduR` (`PDU Router`)
- `EcuC` (`ECU Configuration`)
- `Can` (`Can Driver`)
- `CanIf` (`Can Interface`)
- `Fr` (`FlexRay Driver`)
- `FrIf` (`FlexRay Interface`)
- `Lin` (`LIN Driver`)
- `LinIf` (`LIN Interface`)

Limitations

`Lin` and `LinIf` module configurations can only be imported if they are in Autosar version 2.1.

4.11.2.8.3. Importing TDB configurations

The TDB Importer enables importing into system model and into the ECU configuration of the project. The GUI of the importer allows to enter parameters for both import scenarios.

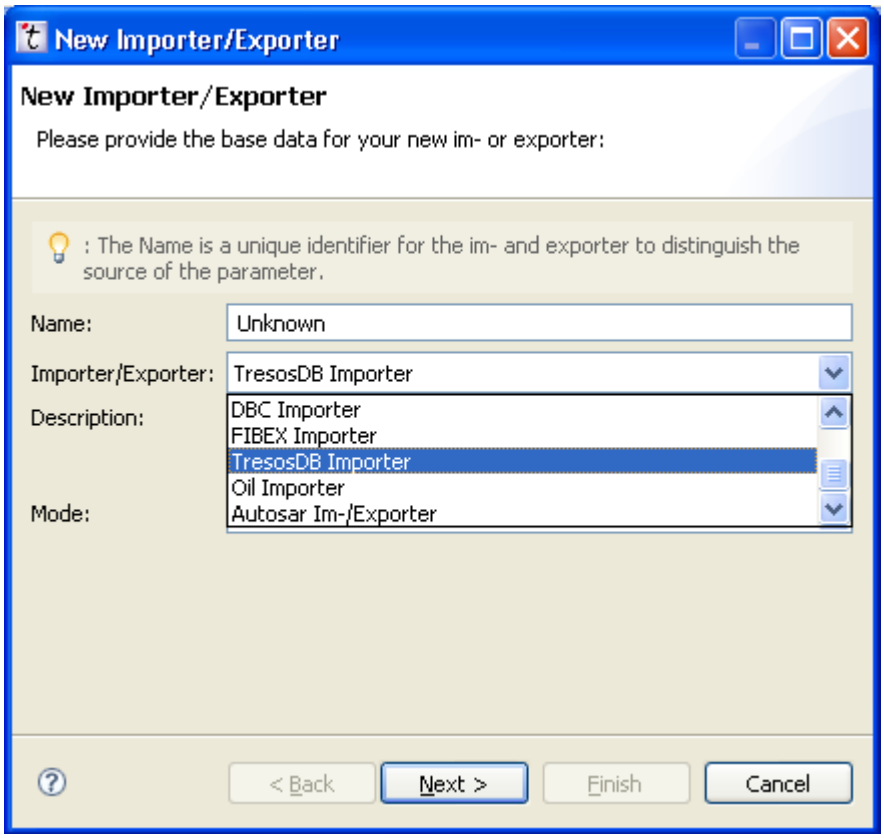
To import configurations from the TDB format:

- Refer to Section 4.11.2: Importing and exporting configuration data.
- Follow the instructions of Step 1 until you arrive at the window *Create, manage and run im- and exporters*.

In the window *Create, manage and run im- and exporters*:

- Select the **Create a new im- or exporter** button.

The *New Importer/Exporter* window opens up.



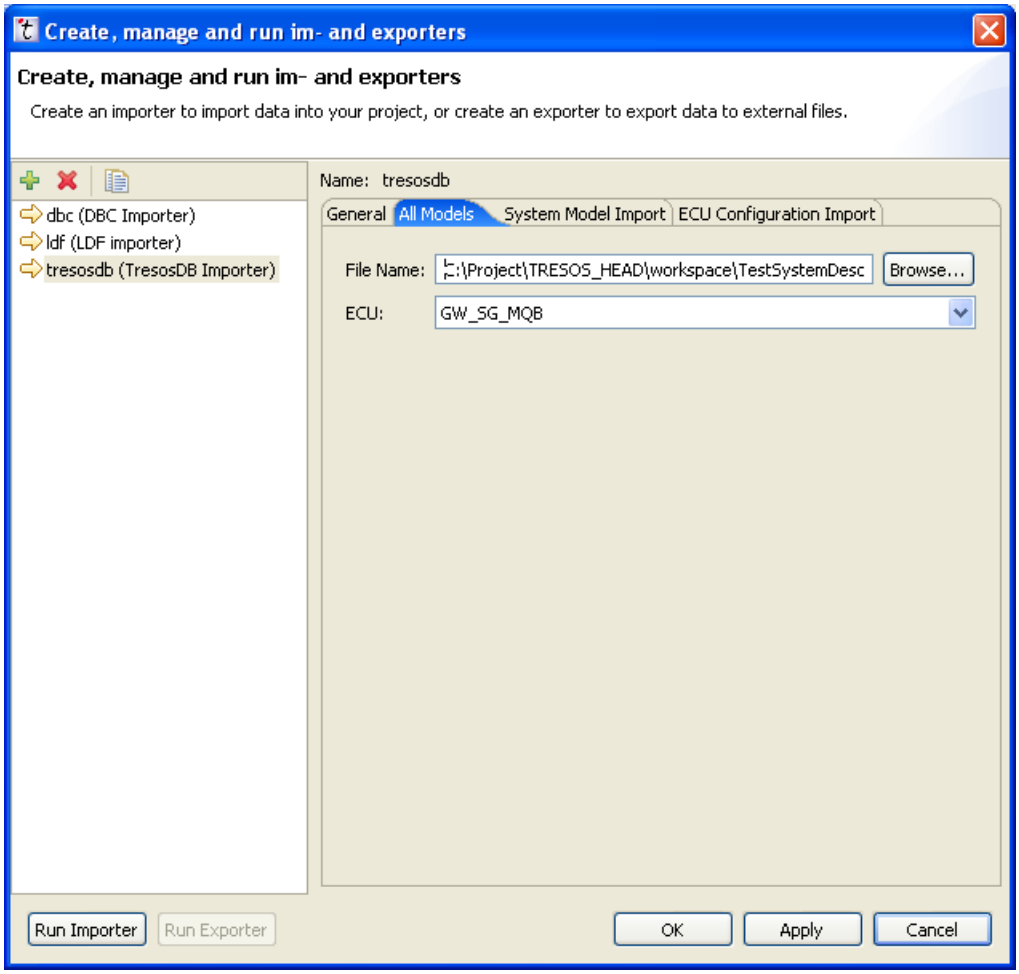
To enter the basic information for the new importer or exporter configuration:

- Select the *Name* textbox.
- Enter a name for your new configuration.
- Select *TDB Importer* from the **Importer/Exporter** dropdown listbox.
- Select **Import only** from the *Mode* dropdown listbox.

The TDB Importer can only be used to import files. Exporting to TDB format is not possible.

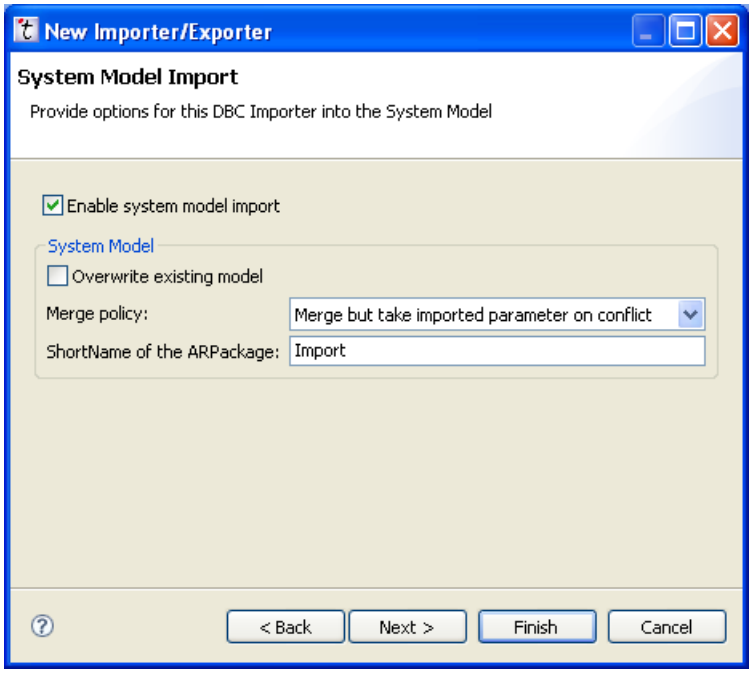
- Select **Next**.

After pressing the **Next** button, you may enter general information.



- Use the **File Name** file selection entry to select the TDB file that you would like to import.
- To import the ECU from the TDB file, select the ECU from the **ECU** dropdown listbox.

Press the **Next** button to switch to a page where parameters for the system model import can be entered.

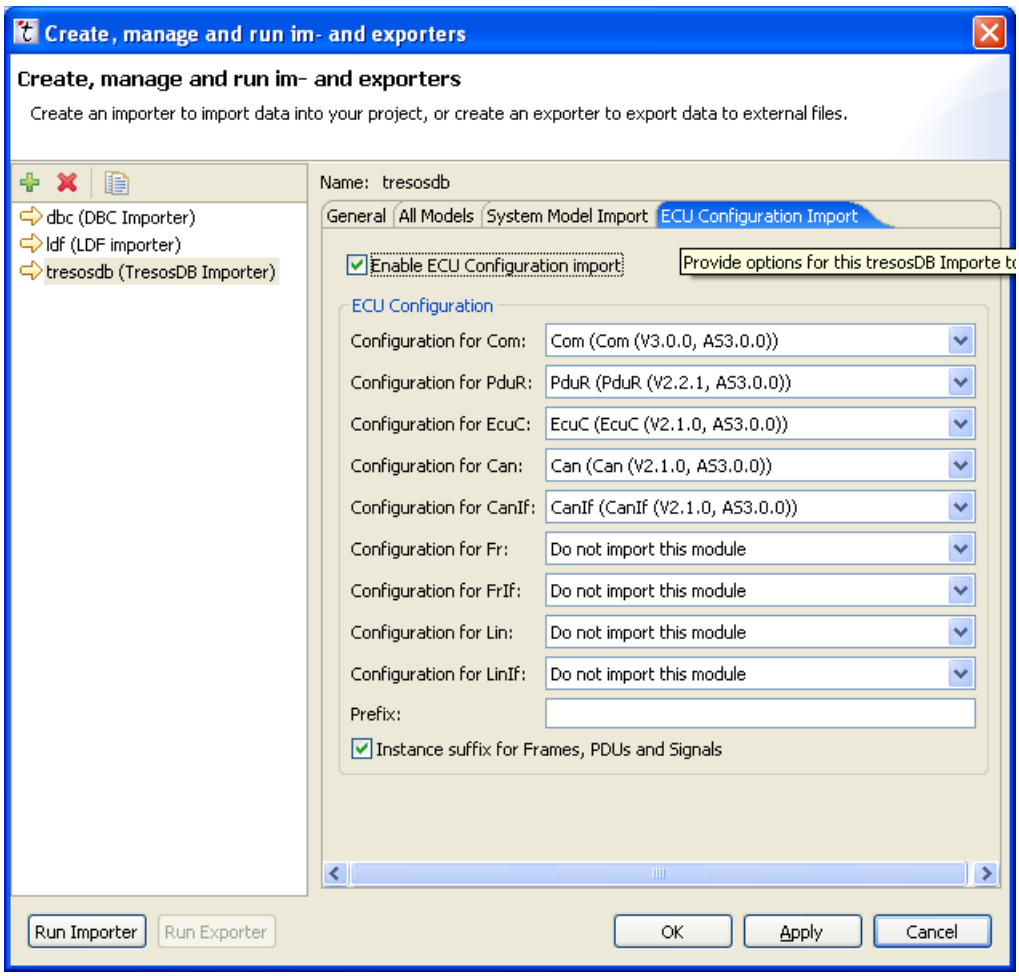


- To enable system model import into the project, check the *Enable system model import* checkbox. If the project has no system model, the complete page including the checkbox is greyed out.
- To completely replace the system model in the project, check the checkbox *Overwrite existing model*.
- To provide a hint for the merge algorithm that merges the imported configuration into the system model of the project, select an entry from the *Merge policy* dropdown listbox .
- To select the Autosar SHORT-NAME of the Autosar AR-PACKAGE into which the system model of the source file is to be imported, enter the Autosar SHORT-NAME into the **ShortName of the ARPackage** textbox.
- Select **Next** to proceed to the next configuration window.

The *ECU Configuration Import* window is provided for entering ECU configuration-specific parameters.

Note

Since the TDB Importer does not create module configurations, you need to assure that a module configuration is present for every module you would like to import into (Com, PduR, Can, CanIf, FrIf, Fr, Lin, LinIf, EcuC).



- To disable ECU configuration import, check the *Enable ECU Configuration Import* checkbox .
- In the *Configuration for Com* dropdown listbox, select the target Com configuration for the import.
- In the *Configuration for PduR* dropdown listbox, select the target PduR configuration for the import.
- In the *Configuration for Can* dropdown listbox, select the target Can configuration for the import.
- In the *Configuration for CanIf* dropdown listbox, select the target Can Interface configuration for the import.
- In the *Configuration for Fr* dropdown listbox, select the target FlexRay configuration for the import.
- In the *Configuration for FrIf* dropdown listbox, select the target FlexRay Interface configuration for the import.
- In the *Configuration for Lin* dropdown listbox, select the target Lin configuration for the import.
- In the *Configuration for LinIf* dropdown listbox, select the target LIN Interface configuration for the import.
- In the *Configuration for EcuC* dropdown listbox, select the target EcuC configuration for the import.
- For details on the *Prefix* textbox see Section 4.11.3.1.2: Name prefixing.
- For details on the *Instance Handling* checkbox see Section 4.11.3.1.3: Instance handling.

To save the importer configuration:

- Select **Finish**.

This saves your importer configuration and closes the *TDB Importer Options* window.

- To run the TDB Importer, select the importer name you just created on the left side of the *Create, manage, and run im- and exporters* window.
- Select **Run Importer**.

The TDB configuration is being imported.

4.11.2.8.4. TDB to Autosar parameter mapping

In case the .tdb file originates from EB tresos Designer, it contains the configuration of a FlexRay network.

- To import FlexRay configurations into EB tresos Studio, refer to: Section: Fibex to Autosar parameter mapping.
- To import CAN configurations, refer to Section: DBC to Autosar parameter mapping .
- To import LIN configurations, refer to Section: LDF to Autosar parameter mapping.

4.11.3. Background information on importing network configuration formats

This section contains some general information that applies to all importers of network configuration formats (e.g. DBC, LDF, Fibex).

4.11.3.1. Naming rules

4.11.3.1.1. Name mangling algorithm for DBC and LDF Importers

In Autosar configuration files container names must be valid C-Identifiers and must have a length of at most 32 characters. The names of the containers must be unique within the context of the parent containers. In order to ensure that all containers created during DBC and LDF import comply to the conditions above, object names from DBC and LDF imports are mangled.

Any characters violating the rules for valid C-Identifiers are replaced by underscores (_). In case a name exceeds 30 characters it is cut off and an underscore followed by a unique index is added so that the resulting name

has a length of at most 30 characters and is unique. Since Autosar allows a length of at most 32 characters, the DBC and LDF Importers can add prefixes which are at most two characters long without violating Autosar naming rules, e.g. `SG` is added to normal signals, `NW` is added to network signals.

Examples:

- `TestName901234567890123456789 0Foo` becomes `TestName9012345678901234567 8_1`
- `Test-Name901234567890123456789 0Bar` becomes `TestName9012345678901234567 8_2`

Note

If the check for the maximum container name length is turned off (see Section 4.5.3.2: Setting Autosar preferences), names are mangled to 110 characters instead of 30.

4.11.3.1.2. Name prefixing

Prior to import a prefix can be defined which will be added to the ShortName of every imported container. The prefix menu line may be left empty if you do not wish a prefix.

The entered prefix will be referred to as `<PREFIX>` in this document.

4.11.3.1.3. Instance handling

In Autosar module configurations, instances of frames, PDUs, and signals are handled in a different way depending on the bus system. Whereas for CAN and LIN configurations, there is usually a one to one relationship between prototype and instance, this is usually not the case for FlexRay configurations. Since every instance container of a configuration needs its unique name, and unique names are usually only provided for prototypes, the problem on how to obtain unique container names for instances of the same prototype arises. Moreover, IPduM configurations allow that multiple instances of the same sub PDU reside in one multiplexed PDU, which also makes it necessary to add MUX information to the instance name.

EB tresos Studio provides an option by which the instance names are generated by appending a suffix to the name of the prototype which guarantees that the instance name will be unique.

For FlexRay frames, the following information is added to instances of frames:

- FlexRay Identifier (1..2047)
- Channel Information (A|B)
- Base Cycle (0..63)
- Cycle Repetition (1|2|4|8|16|32|64)
- Transmission Direction Information (T|R)

An instance of a frame "TestFrame", received in Slot 123, Base Cycle 10, Cycle Repetition 32 on Channel A would therefore be called `TestFrame_123A1032R`.

For CAN frames, the following information is added to instances of frames:

- CAN Identifier (0..2^29-1)
- Transmission Direction Information (T|R)

An instance of a frame "TestFrame", received with CAN Id 123 would therefore be called `TestFrame_123R`.

For LIN frames, the following information is added to instances of frames:

- LIN Identifier (0..63)
- Transmission Direction Information (T|R)

Although a Lin frame/PDU can be sent or received in multiple Schedule Tables or even multiple times in the same Schedule, the layer above only sees one instance of this frame/PDU. Therefore the Schedule Table and the entry position information has been left out in the instance suffix. An instance of a frame `TestFrame`, received with Lin Id `23` would therefore be called `TestFrame_23R`.

The naming scheme for frame instances can also be applied to PDU instances if the PDU is not a sub-PDU of a multipleded PDU. The instance of PDU "TestPdu" which is sent in the frame instance `TestFrame_123A1032R` is named `TestPdu_123A1032R`. If a PDU is a sub-PDU of a multiplexed PDU, additional information must be added:

- Static/Dynamic MUX part prefix S|D
- MUX value for PDUs in the dynamic part (0..255)

If `StaticPdu` is a sub-PDU instance contained in the static part of `TestPdu_123A1032R`, and `DynamicPdu` is contained in the dynamic part, where the MUX Value is `0`, the PDU instances are named `StaticPdu_123A1032RS` and `DynamicPdu_123A1032RD0`.

The naming scheme for signal instances adopts the naming scheme from the IPdu instance in which it is contained. A signal instance of `TestSignal` transmitted in the PDU instance `TestPdu_123A1032R` is called `TestSignal_123A1032R`.

For an import run, add instance suffixes by checking the **Instance suffix for Frames, PDUs and Signals** checkbox.

The option checkbox shoud be turned on if FlexRay networks are imported. For CAN and LIN networks, this option can usually be turned off.

Please also note that the length of the container names grows considerably if the option is turned on, which usually makes it necessary to turn off the check for the maximum container name length. See Section 4.5.3.2: Setting Autosar preferences for instructions on how this check is turned off.

The suffix is referred to by `<INSTSUFFIX>` in this document.

4.11.3.1.4. Signal offsets

Signals can be packed into PDUs either in big endian or little endian format. The endianess information, together with the signal offset parameter defines the bitset within the PDU which is occupied by a signal. Whereas the endianess of a signal within its enclosing PDU is an agreed upon notion, virtually every document format developed its own peculiar definition of the notion signal offset making an implicit offset conversion during import runs necessary.

Please refer to the external document format specification and to the Autosar System Template specification on how the signal offset is defined.

4.12. Working with the command line

EB tresos Studio provides a command line interface to trigger

- code generation and verification for configuration projects (see Section 4.12.2: Commands for project-based code generation and configuration),
- code generation and verification independent from projects (see Section 4.12.3: Commands for legacy code generation),
- module development functions (see Section 4.12.4: Commands for module development), and
- file conversion functions (see Section 4.12.5: Commands for converting and merging of files).

4.12.1. General

4.12.1.1. Starting the command line

To start the commandline:

- Open the Windows command prompt.
- Change into the directory where you have installed EB tresos Studio; e.g. `C:\EB\tresos`.
- Change into the `bin` subdirectory.
- You can execute a command by using `tresos_cmd.bat`; e.g. type `tresos_cmd.bat --help` to see a short help for the commandline options.

4.12.1.2. The command line syntax

An example of a command line is:

```
tresos_cmd.bat -Dinfo=false -data c:\workspace generate myproject
```

The EB tresos Studio command line has a general syntax which is used throughout all available commands. Each command is composed of certain blocks. Some of the blocks may be left out, but the overall order of the blocks is important for the command to be recognized properly. A command line consists of the following blocks:

```
tresos_cmd.bat [<system_property>...] [-data <workspace_dir>]
               <command>
```

The position of these blocks is important.

Block	Description
`<system_property>`	System properties can control the overall behaviour of the command. Some system properties are understood by every command and some are only valid for a certain command. The list of system properties understood by all commands can be found in Section 4.12.1.3: System properties common to all commands. System properties always appear first on the command line. You can specify as many system properties as you want, or non at all.
`<workspace_dir>`	This block lets you control on which workspace the command line operates. You may omit this block. Details of the contents of the block can be found in Section 4.12.1.4: Changing the workspace on the command line.
`<command>`	This block tells EB tresos Studio what to do, e.g. to generate code. This block consists of the command itself and additional parameters that control what the command does. The other sections of this chapter describe all the commands that are available and their parameters.

4.12.1.3. System properties common to all commands

The following system properties can be used on all the commands explained below:

System Property	Description
`-Dinfo=false`	Disable printing of any info messages.
`-Dwarning=false`	Disable printing of any warning messages.

System Property	Description
`-Derror=false`	Disable printing of any error messages.
`-Dmsg<code>=false`	Disable printing of any messages with the given numerical error code `<code>`.
`-Ddebug.packages=<Package>:<Package>:...`	Enable Log4J debugging for specific Java packages.
`-DValidate=true`	Validate the read and written files (e.g. against an Autosar xsd-schema-file).
`-Dbase.path=<directory>`	The base directory for the command.

An example for the use of system properties is:

```
tresos_cmd.bat -Dinfo=false -Dmsg1234=false generate myproject
```

This will generate code for a project called *myproject*. When the commandline is executed, the output will not contain any informational messages (`-Dinfo=false`) and additionally the output of the message with the code *1234* will also be suppressed (`-Dmsg1234=false`).

4.12.1.4. Changing the workspace on the command line

By default EB tresos Studio loads the workspace used in the last program run (see Section 4.2.2: EB tresos Studio projects and workspaces). When executing commands on the commandline you often want EB tresos Studio to use a certain workspace directory and not the one used last. To do so, you can specify a new workspace directory with the `-data` parameter. This parameter changes the workspace directory for this program run only. If you execute another command, or start the GUI, the worskpace used previously is reused. The `-data` option must be placed behind `tresos_cmd.bat`, after any system properties, and takes a directory as argument.

An example for how the workspace can be changed is:

```
tresos_cmd.bat -Dinfo=false -data c:\workspace generate myproject
```

This will generate code for a project called *myproject*. As workspace, the directory `c:\workspace` is used. This implies that the project *myproject* is part of that workspace. When the commandline is executed, the output will not contain any informational messages.

Note

The system property `-Dinfo=false` can also be ommitted. It is only present to demonstrate the correct order of the commandline parameters.

The following example does the same as above, but uses the workspace which was used last:

```
tresos_cmd.bat -Dinfo=false generate myproject
```

4.12.2. Commands for project-based code generation and configuration

Use the project-based mode of the command line to generate code, verify configurations, or to change the configuration for an existing EB tresos Studio project stored in an EB tresos Studio workspace (see Section 4.2.2: EB tresos Studio projects and workspaces).

4.12.2.1. Verifying a project

To verify the configuration of all modules and to print out all warnings and errors, use the `verify` command.

4.12.2.1.1. Syntax

```
tresos_cmd.bat [<system_property>...] [-data <workspace>]
               verify <project>
```

4.12.2.1.2. System properties

This command does only recognize the system properties common to all commands (see Section 4.12.1.3: System properties common to all commands).

4.12.2.1.3. Parameters

In addition to the common parameters `<system_property>` and `<workspace>`, this command does also recognize the following parameters:

Parameter	Description
`<project>`	The name of the configuration project on which you want the command to run.

4.12.2.1.4. Example

The following code is an example of how to use the command `verify`:

```
tresos_cmd.bat -Dinfo=false -Dwarning=false verify myproject
```

This command verifies the configuration of the project *myproject* and does not write warning and info messages as output.

4.12.2.2. Generating code for a project

To verify the configuration and to generate code for the modules of a configuration project and to print out all warnings and errors, use the `generate` command.

4.12.2.2.1. Syntax

```
tresos_cmd.bat [<system_property>...] [-data <workspace>]
               generate [<option>...] <project>
```

4.12.2.2.2. System properties

This command does only recognize the system properties common to all commands (see Section 4.12.1.3: System properties common to all commands).

4.12.2.2.3. Options

The following options are supported by this command:

Parameter	Description
`-o <output_dir>`	By default, the same output directory as in the GUI is used. If you want to generate you code into a different directory on the command line, use this option and replace `<output_dir>` by the directory you want to generate the code to.
`-g <module-id>`	By default, the modules which are enabled for generation in the GUI will be generated (see Section 4.7.5: Disabling code generation of a module configuration). You can use the `-g` option to override that setting and to generate only the modules you specify on this command line. You can use the `-g` option

Parameter	Description
	mupltiple times if you want to generate code for more than one module. The argument `<module-id>` must be replaced by the ID of the module you want to generate code for, e.g. `Com_TS_T16D4M2I0R0`. By default, only the module configurations for which code is being generated will be verified. To verify all module configurations, even if no code is generated for them, use the `-a` option. **Note** You can only specify module IDs of modules that are part of your project.
`-a`	If you are using the `-g` option, only the configurations of the modules specified will be verified before the code is generated. If you want all module configurations to be verified, even if no code is generated for them, use this option.

4.12.2.2.4. Parameters

In addition to the common parameters `<system_property>` and `<workspace>`, this command does also recognize the following parameters:

Parameter	Description
`<project>`	The name of the configuration project on which you want the command to run.

4.12.2.2.5. Example

The following code is an example of how to use the command `generate`:

```
tresos_cmd.bat -Dinfo=false -Dwarning=false generate myproject
```

This command generates code for the modules of the configuration project *myproject* and does not write warning and info messages as output.

4.12.2.3. Listing custom-built generation modes

Some modules may define additional generator modes which can be run with the `make` command (see Section 4.12.2.4: Executing custom-built generation modes for projects). If the generator is available, depends on

whether the module providing that generator is part of your project. To get a list of all supported generators of a given project.

4.12.2.3.1. Syntax

```
tresos_cmd.bat [<system_property>...] [-data <workspace>]
               listmodes <project>
```

Tip

Some of the EB tresos AutoCore modules define additional generator modes. If you purchased these modules, please refer to that documentation to learn more about supported modes.

4.12.2.3.2. System properties

This command does only recognize the system properties common to all commands (see Section 4.12.1.3: System properties common to all commands).

4.12.2.3.3. Parameters

In addition to the common parameters `<system_property>` and `<workspace>`, this command does also recognize the following parameters:

Parameter	Description
`<project>`	The name of the configuration project on which you want the command to run.

4.12.2.3.4. Example

The following code is an example of how to use the command `listmodes`:

```
tresos_cmd.bat listmodes myproject
```

This command lists all the available modes for the configuration project *myproject* and writes them to the command line.

4.12.2.4. Executing custom-built generation modes for projects

It is possible for a module to provide a custom-built generator. This generator then can be run by you by calling a new generator mode, other than generate and verify. The name of the generator mode and what it does depends on the module. To find out if a module contributes such a custom mode, please refer to the documentation of the modules you are running with EB tresos Studio.

4.12.2.4.1. Syntax

If a module is part of your project that provides a custom-built generator, you can run it with the following command line:

```
tresos_cmd.bat [<system_property>...] [-data <workspace>]
               make <mode> <project>
```

4.12.2.4.2. System properties

By default, this command does only recognize the system properties common to all commands (see Section 4.12.1.3: System properties common to all commands). But it is possible for custom-built generators to recognize other system properties. Please refer to the documentation of the generator you are planning to use, for more information.

4.12.2.4.3. Parameters

In addition to the common parameters `<system_property>` and `<workspace>`, this command does also recognize the following parameters:

Parameter	Description
`<mode>`	The name of the custom-built generator. Please refer to the documentation of the generator you are planning to use to find out more about the usage. **Note** To use a custom-built generator on a project, the module which provides that generator must be part of the project.

Parameter	Description
	Tip You can use the `<listmodes>` command (see Section 4.12.2.3: Listing custom-built generation modes) to find out which generator modes are available.
`<project>`	The name of the configuration project on which you want the command to run.

4.12.2.4.4. Example

The following code is an example of how to use the command `make`:

```
tresos_cmd.bat make generate_SWC-T myproject
```

This command runs a custom-built generator called *generate_SWC-T* on the project *myproject*.

4.12.2.5. Executing autoconfigure wizards on projects

If you have created a configuration for an autoconfigure wizard of a project in the GUI, you can execute this wizard on the commandline. For more information about autoconfigure wizard, please refer to Section 4.9.3: Autoconfigure wizards.

4.12.2.5.1. Syntax

To execute an autoconfigure wizard:

```
tresos_cmd.bat [<system_property>...] [-data <workspace>]
               autoconfigure <project> [<wizard>]...
```

4.12.2.5.2. System properties

This command does only recognize the system properties common to all commands (see Section 4.12.1.3: System properties common to all commands).

4.12.2.5.3. Parameters

In addition to the common parameters `<system_property>` and `<workspace>`, this command does also recognize the following parameters:

Parameter	Description
`<project>`	The name of the configuration project on which you want the autoconfigure wizard to run.
`<wizard>`	The name of the autoconfigure wizard that shall be run. You can run multiple wizards in a row by providing several names separated by spaces. If you do not specify any wizards, all configured wizards are executed.

4.12.2.5.4. Example

The following code is an example of how to use the command `autoconfigure`:

```
tresos_cmd.bat autoconfigure myproject
```

This command runs all autoconfigure wizards that are configured and checked in the GUI on the project *myproject*. The behaviour of these wizards is controlled by the settings provided in the GUI.

4.12.2.5.5. Running the Calculate Default Values wizard on the command line

To run the Calculate Default Values wizard as configured in the GUI on a project called *myproject*, use:

```
tresos_cmd.bat autoconfigure myproject AutoCalc
```

4.12.2.5.6. Running the Calculate Handle-Ids wizard on the command line

To run the Calculate Handle-Ids wizard as configured in the GUI on a project called *myproject*, use:

```
tresos_cmd.bat autoconfigure myproject HandleId
```

4.12.3. Commands for legacy code generation

In addition to the project-based mode, EB tresos Studio does also support a so-called legacy-mode. This mode works independent from projects. This section describes all legacy-mode commands that handle the generation of code. There are more legacy-mode commands which will be explained in later sections.

Tip

Specifying a workspace location in legacy-mode does not have much effect, because there are no projects in legacy mode. However, the workspace is still used internally, e.g. as location for the errorlog file.

4.12.3.1. Naming conventions for modules usable in legacy-mode

If a module configuration is used in a legacy mode command, the module configuration first is loaded from the given file. Then, the tool tries to locate the corresponding module. Since the legacy mode works independent from projects and from plugins, the modules must conform to a certain naming convention in order to be found.

EB tresos Studio uses the following naming convention to identify a module:

```
Module-name = <MODULE-CONFIGURATION>_<AR-PACKAGE>
```

- `<MODULE-CONFIGURATION>` stands for the name of the module configuration.
- `<AR-PACKAGE>` stands for the name of the Autosar package.

Example: the module configuration with the name `Com` is located in AR-PACKAGE `TS_T16D4M2I0R0`, so the module name is: `Com_TS_T16D4M2I0R0`.

Note

If you specify modules with the `-g` option in your command, this naming convention does not apply and you can use any name for the module.

4.12.3.2. Listing all modules available for legacy-mode

4.12.3.2.1. Syntax

To print out the names of all installed modules that support legacy code generation, use the `legacy list` command:

```
tresos_cmd.bat [<system_property>...] [-data <workspace>] legacy list
```

4.12.3.2.2. System properties

This command does only recognize the system properties common to all commands (see Section 4.12.1.3: System properties common to all commands).

4.12.3.2.3. Example

The following is an example of how to use the command `legacy list`:

```
tresos_cmd.bat legacy list
```

4.12.3.3. Generating code for legacy modules

4.12.3.3.1. Syntax

Use the `legacy generate` command to verify module confiogurations and, if no errors took place, generate code for them:

```
tresos_cmd.bat [<system_property>...] [-data <workspace>]
               legacy generate [<options>]
               <config-file>[@<type>]...
```

4.12.3.3.2. System properties

In addition to the common system properties (Section 4.12.1.3: System properties common to all commands), this command does also recognize the following system properties:

System Property	Description
`-DmergeConfigs=(true|false)`	If set to `true`, the parameters of the module configurations will be merged. This is useful, if you have split the parameters of a module configuration into several files (e.g. one file containing standard- and one containing vendor-specific parameters).

System Property	Description
	If this property is set to `false`, no merge of the parameters is done. Instead, the content of the file read last overwrites the previous content. The default is `false` for this command.
`-DverifyReadOnly=(true|false)`	Enable/disable the verification of readonly configuration parameters. The default is `false` for this command.
`-DsuppressReadOnlyWarnings=(true|false)`	If set to `true`, warnings regarding readonly configuration parameters are not logged or printed out. The default is `false` for this command.

4.12.3.3.3. Options

The following options are supported by this command:

Parameter	Description
`-a`	If you are using the `-g` option, only the configurations of the modules specified will be verified before the code is generated. If you want all module configurations to be verified, even if no code is generated for them, use this option.
`-c <config-path>`	If this option is present, the module configuration files will be searched in the directory specified as `<config-path>`. By default, the directory `<base.path>\config` is used, with `base.path` being the value of the system property `-Dbase.path` (see Section 4.12.1.3: System properties common to all commands). If the system property is not set, an error is displayed.
`-d<var>=<value>`	Defines a variable named `<var>` with value `<value>` which then is available to the code generators. If a code generator for a module supports this feature, its documentation should list the available variables and allowed values.
`-g <module-id>`	By default, the code for all given module-configurations, will be generated. You can use the `-g` option to override that setting and to generate code for only a subset of these modules. You can use this option mupltiple times if you want to generate code for more than one module. The argument `<module-id>` must be replaced by the ID of the module you want to generate code for, e.g. `Com_TS_T16D4M2I0R0`. By default, only the module configurations for which code is being generated will be verified. To verify all module configurations, even if no code is generated for them, use the `-a` option. **Note** You can only specify module IDs of modules that are installed in the copy of EB tresos Studio you are running.

Parameter	Description
`-n <config-name>`	Name of the module configurations of the config files to load. If this option is used, other module configurations contained in the specified files will be ignored. You can use this option multiple times.
`-o <output-path>`	If this option is present, the generated files will be written to the directory specified as `<output-path>`. By default, the directory `<base.path>\output` is used, with `base.path` being the value of the system property `-Dbase.path` (see Section 4.12.1.3: System properties common to all commands). If the system property is not set, an error is displayed.
`-u`	If this option is present, the generated files will contain Unix line endings.

4.12.3.3.4. Parameters

In addition to the common parameters `<system_property>` and `<workspace>`, this command does also recognize the following parameters:

Parameter	Description
`<config-file>[@<type>]`	Use `<config-file>` for the name of the file containing the module configuration(s) that shall be loaded. The parameter can be passed multiple times to load more than one file. The optional argument `@<type>` can be used to explicitly specify the content type of the file. See Section 4.2.2.3: File content types for more information on what content types are available.

4.12.3.3.5. Example

The following code is an example of how to use the command `legacy generate`:

```
tresos_cmd.bat -DValidate=true legacy generate -n Com -g
               Com_TS_T16D4M2I0R0 -g Custom -u All.epc@asc:2.0
```

This example loads a module configuration named *Com* from the file *All.epc*. If this file contains other module configurations as well, they will be ignored. The .epc file is treated as Autosar 2.0 file. The module definition for the configuration may be contained in the module *Com_T16D4M2I0R0*, or it will be found by naming convention (see Section 4.12.3.1: Naming conventions for modules usable in legacy-mode). The configuration will be validated against its definition. The code is generated for the two modules *Com_T16D4M2I0R0* and *Custom* (provided that the module *Custom* does not need a configuration). The generated files will contain Unix linefeeds.

4.12.3.4. Verifying configurations for legacy modules

Use the `legacy verify` command to verify module configurations and to print out all infos, warnings, and errors. This command executes the same generation-steps as the `legacy generate` command with the only difference that no files will be written to disk.

4.12.3.4.1. Syntax

```
tresos_cmd.bat [<system_property>...] [-data <workspace>]
               legacy verify [<options>]
               <config-file>[@<type>]...
```

4.12.3.4.2. System properties

In addition to the common system properties (Section 4.12.1.3: System properties common to all commands), this command does also recognize the following system properties:

System Property	Description
`-DmergeConfigs=(true|false)`	If set to `true`, the parameters of the module configurations will be merged. This is useful, if you have split the parameters of a module configuration into several files (e.g. one file containing standard- and one containing vendor-specific parameters). If this property is set to `false`, no merge of the parameters is done. Instead, the content of the file read last overwrites the previous content. The default is `false` for this command.
`-DverifyReadOnly=(true|false)`	Enable/disable the verification of readonly configuration parameters. The default is `false` for this command.
`-DsuppressReadOnlyWarnings=(true|false)`	If set to `true`, warnings regarding readonly configuration parameters are not logged or printed out. The default is `false` for this command.

4.12.3.4.3. Options

The following options are supported by this command:

Parameter	Description
`-c <config-path>`	If this option is present, the module configuration files will be searched in the directory specified as `<config-path>`. By default, the directory `<base.path>\config` is used, with `base.path` being the value of the system property `-Dbase.path` (see Section 4.12.1.3: System properties common to all commands). If the system property is not set, an error is displayed.
`-d<var>=<value>`	Defines a variable named `<var>` with value `<value>` which then is available to the code generators. If a code generator for a module supports this feature, its documentation should list the available variables and allowed values.
`-g <module-id>`	By default, the code for all given module-configurations, will be verified. You can use the `-g` option to override that setting and to verified code for only a subset of these modules. You can use the `-g` option mupltiple times if you want to verified code for more than one module. The argument `<module-id>` must be replaced by the ID of the module you want to verified code for, e.g. `Com_TS_T16D4M2I0R0`. **Note** You can only specify module IDs of modules that installed in the copy of EB tresos Studio you are running.
`-n <config-name>`	Name of the module configurations of the config files to load. If this option is used, other module configurations contained in the specified files will be ignored. You can use this option multiple times.

4.12.3.4.4. Parameters

In addition to the common parameters `<system_property>` and `<workspace>`, this command does also recognize the following parameters:

Parameter	Description
`<config-file>[@<type>]`	Use `<config-file>` for the name of the file containing the module configuration(s) that shall be loaded. The parameter can be passed multiple times to load more than one file. The optional argument `@<type>` can be used to explicitly specify the content type of the file. See Section 4.2.2.3: File content types for more information on what content types are available.

4.12.3.4.5. Example

The following code is an example of how to use the command `legacy generate`:

```
tresos_cmd.bat -DValidate=true legacy validate -n Com -g
               Com_TS_T16D4M2I0R0 -g Custom -u All.epc@asc:2.0
```

This example loads a module configuration named *Com* from the file *All.epc*. If this file contains other module configurations as well, they will be ignored. The .epc file is treated as Autosar 2.0 file. The module definition for the configuration may be contained in the module *Com_T16D4M2I0R0*, or it will be found by naming convention (see Section 4.12.3.1: Naming conventions for modules usable in legacy-mode). The configuration will be validated against its definition.

4.12.3.5. Listing all custom-built generation modes for legacy modules

It is possible for a module to provide a custom-built generator mode. This generator then can be run by you by calling a new generator mode, other than generate and verify. The name of the generator mode and what it does depends on the module. To find out if a module adds such a custom mode, please refer to the documentation of the modules you are running with EB tresos Studio.

4.12.3.5.1. Syntax

To list all available legacy generation modes for custom-built generators of all modules in legacy mode:

```
tresos_cmd.bat [<system_property>...] [-data <workspace>]
               legacy listmodes
```

4.12.3.5.2. System properties

This command does only recognize the system properties common to all commands (see Section 4.12.1.3: System properties common to all commands).

4.12.3.5.3. Example

The following is an example of how to use the command `legacy listmodes`:

```
tresos_cmd.bat legacy listmodes
```

If you run this command you will see all module IDs and their available generation modes.

4.12.3.6. Running custom-built generation modes for legacy modules

It is possible for a module to provide a custom-built generator mode. This generator then can be run by you by calling a new generator mode, other than generate and verify. The name of the generator mode and what it does depends on the module. To find out if a module adds such a custom mode, please refer to the documentation of the modules you are running with EB tresos Studio.

4.12.3.6.1. Syntax

If a module supports the legacy mode, you can run its custom-built generator on a configuration with the following command line:

```
tresos_cmd.bat [<system_property>...] [-data <workspace>]
               legacy make <mode> [<options>]
               <config-file>[@<type>]...
```

4.12.3.6.2. System properties

By default, this command does only recognize the system properties common to all commands (see Section 4.12.1.3: System properties common to all commands). But it is possible for custom-built generators to recognize other system properties. Please refer to the documentation of the generator you are planning to use, for more information.

4.12.3.6.3. Options

The following options are supported by this command:

Parameter	Description
`-a`	If you are using the `-g` option, only the configurations of the modules specified will be verified before the code is generated. If you want all module configurations to be verified, even if no code is generated for them, use this option.

Parameter	Description
`-c <config-path>`	If this option is present, the module configuration files will be searched in the directory specified as `<config-path>`. By default, the directory `<base.path>\config` is used, with `base.path` being the value of the system property `-Dbase.path` (see Section 4.12.1.3: System properties common to all commands). If the system property is not set, an error is displayed.
`-d<var>=<value>`	Defines a variable named `<var>` with value `<value>` which then is available to the code generators. If a code generator for a module supports this feature, its documentation should list the available variables and allowed values.
`-g <module-id>`	By default, the code for all given module-configurations, will be generated. You can use the `-g` option to override that setting and to generate code for only a subset of these modules. You can use the `-g` option mupltiple times if you want to generate code for more than one module. The argument `<module-id>` must be replaced by the ID of the module you want to generate code for, e.g. `Com_TS_T16D4M2I0R0`. By default, only the module configurations for which code is being generated will be verified. To verify all module configurations, even if no code is generated for them, use the `-a` option. **Note** You can only specify module IDs of modules that installed in the copy of EB tresos Studio you are running.
`-n <config-name>`	Name of the module configurations of the config files to load. If this option is used, other module configurations contained in the specified files will be ignored. You can use this option multiple times.
`-o <output-path>`	If this option is present, the generated files will be written to the directory specified as `<output-path>`. By default, the directory `<base.path>\output` is used, with `base.path` being the value of the system property `-Dbase.path` (see Section 4.12.1.3: System properties common to all commands). If the system property is not set, an error is displayed.
`-u`	If this option is present, the generated files will contain Unix line endings.

4.12.3.6.4. Parameters

In addition to the common parameters `<system_property>` and `<workspace>`, this command does also recognize the following parameters:

Parameter	Description
`<config-file>[@<type>]`	Use `<config-file>` for the name of the file containing the module configuration(s) that shall be loaded. The parameter can be passed multiple times to load more than one file. The optional argument `@<type>` can be used to explicitly specify the content type of the file. See Section 4.2.2.3: File content types for more information on what content types are available.

4.12.3.6.5. Example

The following code is an example of how to use the command `legacy make`:

```
tresos_cmd.bat -DmergeConfigs=true legacy make generate_SWC-T
               myEpc.epc myOther.epc
```

This example loads the content of the two files *myEpc.epc* and *myOther.epc*. The module-configurations contained in *myEpc.epc* are merged with the ones in *myOther.epc*, not overwritten. A custom-built generator called *generate_SWC-T* is invoked. It is assumed that the modules of the configurations contained in the loaded files support this generation mode.

4.12.4. Commands for module development

The following commands are for module developers or debugging purposes mostly. Topics such as printing out data models or creating and checking vendor-specific module definitions are covered.

4.12.4.1. Listing a data model

Use the `legacy tree` command to print out the data model loaded from modules and module configurations.

4.12.4.1.1. Syntax

```
tresos_cmd.bat [<system_property>...] [-data <workspace>]
               legacy tree [<options>] <config-file>[@<type>]...
```

4.12.4.1.2. System properties

In addition to the common system properties (see Section 4.12.1.3: System properties common to all commands), this command does also recognize the following system properties:

System Property	Description
`-DmergeConfigs=(true|false)`	If set to `true`, the parameters of the module configurations will be merged. This is useful, if you have split the parameters of a module configuration into several files (e.g. one file containing standard- and one containing vendor-specific parameters). If this property is set to `false`, no merge of the parameters is done. Instead, the content of the file read last overwrites the previous content. The default is `false` for this command.

4.12.4.1.3. Options

The following options are supported by this command:

Parameter	Description
`-c <config-path>`	If this option is present, the module configuration files will be searched in the directory specified as `<config-path>`. By default, the directory `<base.path>\config` is used, with `base.path` being the value of the system property `-Dbase.path` (see Section 4.12.1.3: System properties common to all commands). If the system property is not set, an error is displayed.
`-n <config-name>`	Name of the module configurations of the config files to load. If this option is used, other module configurations contained in the specified files will be ignored. You can use this option multiple times.

4.12.4.1.4. Parameters

In addition to the common parameters `<system_property>` and `<workspace>`, this command does also recognize the following parameters:

Parameter	Description
`<config-file>[@<type>]`	Use `<config-file>` for the name of the file containing the module configuration(s) that shall be loaded. The parameter can be passed multiple times to load more than one file. The optional argument `@<type>` can be used to explicitly specify the content

Parameter	Description
	type of the file. See Section 4.2.2.3: File content types for more information on what content types are available.

4.12.4.1.5. Example

The following code is an example of how to use the command `legacy generate`:

```
tresos_cmd.bat legacy tree -n Com All.epc@asc:2.0
```

This example loads a module configuration named *Com* from the file *All.epc* into the data model and prints the model as a tree. If this file contains other module configurations as well, they will be ignored. The .epc file is treated as Autosar 2.0 file.

4.12.4.2. Creating vendor-specific module definitions (VSMD)

According to the Autosar specification, a vendor has to derive a vendor-specific module definition (*VsMD*) from the corresponding Autosar standard module definition (*StMD*). You can use the `vsmd` command to let EB tresos Studio derive a basic vendor-specific module definition for you.

Tip

Since it is possible to specify different types for StMD and VsMD, you may also use the `vsmd` command to convert files from one format to the other.

Several rules must be followed for deriving a VsMD from a StMD. These rules are part of the Autosar specification. The most important one requires you to choose an own package name under which your module definition is located. If you use this command, it will create a reference to the StMD from which the VsMD with the `REFINED-MODULE.DEF` tag. In addition, new `UUID`s are created for each element of the VsMD.

4.12.4.2.1. Syntax

Use the `vsmd` command to create a basic vendor-specific module definition from a standard module definition:

```
tresos_cmd.bat [<system_property>...] [-data <workspace>]
               legacy vsmd [<options>] <stmd-file>[@<type>]...
               <vsmd-file>[@<type>] <package> <origin>
```

4.12.4.2.2. System properties

In addition to the common system properties (see Section 4.12.1.3: System properties common to all commands), this command does also recognize the following system properties:

System Property	Description
`-DmergeConfigs=(true|false)`	If set to `true`, the parameters of the module configurations will be merged. This is useful, if you have split the parameters of a module configuration into several files (e.g. one file containing standard- and one containing vendor-specific parameters). If this property is set to `false`, no merge of the parameters is done. Instead, the content of the file read last overwrites the previous content. The default is `true` for this command.
`-DwriteDefaults=(true|false)`	Deprecated property. Can be used to force the `LOWER-MULTIPLICITY` and `UPPER-MULTIPLICITY` tags to be printed to the output-file even if their value is "1". The default value for this parameter is `false` and thus the tags are usually omitted. This property was used in pervious versions to achieve conformance to xsd-schemafiles what is now replaced by content types (see Section 4.2.2.3: File content types)
`-Duuids=(true|false)`	If set to `true`, new random unique IDs are created for each node. The default is `true`.
`-DecuParamDef=(true|false)`	If set to `false`, no `ECU-PARAMETER-DEFINITION` tag is generated into the output file. The default is `true`.
`-DecuParamDefName=<name>`	Changes the name of the `ECU-PARAMETER-DEFINITION` tag in the output file. The default value is `myEcuParameterDefinition`.
`-DbswModuleDesc=(true|false)`	If set to `false`, no `BSW-MODULE-DESCRIPTION` tag is generated into the output file. The default is `true`.
`-DbswModuleDescName=<name>`	Changes the name of the `BSW-MODULE-DESCRIPTION` tag in the output file. The default value is `myBswModuleDescription`.
`-DvsmdPath=<as-path>`	Only useable if the property `-DbswModuleDesc=true` is used. You can use the `<as-path>` argument to explicitly set the value of the `VENDOR-SPECIFIC-MODULE-DEF-REF` parameter inside the `BSW-MODULE-DESCRIPTION` container of the output file. By default, this path is determined automatically.
`-DrecommendedConfigPath=<as-path>`	Only useable if the property `-DbswModuleDesc=true` is used. You can use the `<as-path>` argument to explicitly set the value of the `RECOMMENDED-CONFIGURATION-REF` parameter inside the `BSW-MODULE-DESCRIPTION` container of the output file. If this property is not set, no recommended configuration is loaded.

System Property	Description
`-DpreConfigPath=<as-path>`	Only useable if the property `-DbswModuleDesc=true` is used. You can use the `<as-path>` argument to explicitly set the value of the `PRECONFIGURED-CONFIGURATION-REF` parameter inside the `BSW-MODULE-DESCRIPTION` container of the output file. If this property is not set, no pre-configuration is loaded.
`-DsuppressReadOnlyWarnings=(true|false)`	If set to `true`, warnings regarding readonly configuration parameters are not logged or printed out. The default is `false` for this command.
`-DMapOptionalAsList=(true|false)`	This parameter is only usable if the content type of the output file is `xdm`. If set to `true`, a surrounding list for elements with LOWER-MULTIPLICITY= 0 and UPPER-MULTIPLICITY=1 is created. If set to `false`, no surrounding lists will be created. Instead, the affected elements be marked as optional. For backward-compatibility reasons, the default value is `true` **Warning** Leaving this property at the default, or explicitly setting it to `true` can lead to problems! This makes the XPaths of the affected elements depend on their LOWER-MULTIPLICITY value. If the LOWER-MULTIPLICITY is 0, a list is created. If the LOWER-MULTIPLICITY is later changed to 1, no list is created (with the UPPER-MULTIPLICITY always being at 1). The XPath is different in both cases. Code templates which then wish to address this parameter in the configuration need to be able to handle both cases. On the other hand, if you set this system property to `false`, the XPath stays the same in any case. **Tip** If you are writing a code-template, you can use the following XPath function in the template to determine if the parameter is configured or not: `node:exists(OptionalVariable)`.

4.12.4.2.3. Options

The following options are supported by this command:

Parameter	Description
-c <config-path>	If this option is present, the module configuration files will be searched in the directory specified as <config-path>. By default, the directory <base.path>\config is used, with base.path being the value of the system property -Dbase.path (see Section 4.12.1.3: System properties common to all commands). If the system property is not set, an error is displayed.
-o <output-path>	If this option is present, the generated files will be written to the directory specified as <output-path>. By default, the directory <base.path>\output is used, with base.path being the value of the system property -Dbase.path (see Section 4.12.1.3: System properties common to all commands). If the system property is not set, an error is displayed.
-u	If this option is present, the generated files will contain Unix line endings.

4.12.4.2.4. Parameters

In addition to the common parameters <system_property> and <workspace>, this command does also recognize the following parameters:

Parameter	Description
<stmd-file>[@<type>]	The file name containing the standardized module definition (StMD) from which to create the vendor-specific module definition (VsMD). If the file contains more than one module definitions, all will be loaded. The optional argument @<type> can be used to explicitly specify the content type of the file. See Section 4.2.2.3: File content types for more information on what content types are available.
<vsmd-file>[@<type>]	The file name of the file to which to write the created VsMD. The optional argument @<type> can be used to explicitly specify the content type of the file. See Section 4.2.2.3: File content types for more information on what content types are available. If you do not specify any content type, the file extension is interpreted. Possible extensions are: *.xdm , *.arxml, *.asc, *.epd, *.bmd, *.epc.
<package>	The name of the Autosar package under which your module definition is located.
<origin>	Origin of the parameters. This parameter is only used if the content type of the output file is xdm. In this case, each parameter in the xdm file get an attribute named __ORIGIN with the text that you specified here as value.

4.12.4.2.5. Example

The following code is an example of how to use the command `legacy vsmd`:

```
tresos_cmd.bat -DMapOptionalAsList=false legacy vsmd
               Com.epd MyCom.xdm TS_T16D4M2I0R0
```

This example creates a VsMD file with the name *MyCom.xdm* out of the StMD *Com.epd*. The created VsMD contains the AR-PACKAGE *TS_T16D4M2I0R0*. As recommended, optional elements are not surrounded by lists in the output file.

4.12.4.3. Checking vendor-specific against standard module definitions

There are some rules, you must follow when you derive a vendor-specific module definition (VSMD) from a standardized module definition (StMD).

4.12.4.3.1. Syntax

To check if a VsMD conforms to a StMD, use the `legacy vsmdcheck` command:

```
tresos_cmd.bat [<system_property>...] [-data <workspace>]
               legacy vsmdcheck
               <stmd-file>[@<type>] <vsmd-file>[<type>]...
               <rule|ruleset>[,<rule|ruleset>...]
               [-ignore <rule>]... [-xml <xmlfile>]
```

4.12.4.3.2. System properties

In addition to the common system properties (see Section 4.12.1.3: System properties common to all commands), this command does also recognize the following system properties:

System Property	Description
`-DsuppressReadOnlyWarnings=(true|false)`	If set to `true`, warnings regarding readonly configuration parameters are not logged or printed out. The default is `false` for this command.

4.12.4.3.3. Parameters

In addition to the common parameters `<system_property>` and `<workspace>`, this command does also recognize the following parameters:

Parameter	Description
`<stmd-file>[@<type>]`	The file name containing the standardized module definition (StMD). If the file contains more than one module definitions, all will be loaded. The optional argument `@<type>` can be used to explicitly specify the content type of the file. See Section 4.2.2.3: File content types for more information on what content types are available.
`<vsmd-file>[@<type>]`	The file name containing the vendor-specific module definition (VsMD). If the file contains more than one module definitions, all will be loaded. The optional argument `@<type>` can be used to explicitly specify the content type of the file. See Section 4.2.2.3: File content types for more information on what content types are available.
`<rule|ruleset>`	The rule or rule-set which should be used. For a list of available rules, see the `legacy listrules` command Section 4.12.4.4: Listing all rules used for VSMD checking. Available rulesets are: `asc:2.0`, `asc:2.1` and `asc:3.0`.
`-ignore <rule>`	The name of a rule which shall not be checked. For a list of available rules, see the `legacy listrules` command Section 4.12.4.4: Listing all rules used for VSMD checking.
`-xml <xmlfile>`	The name of a file into which the result shall be saved in xml-format.

4.12.4.3.4. Example

The following code is an example of how to use the command `legacy vsmd`:

```
tresos_cmd.bat -DValidate=true legacy vsmdcheck StMD.bmd@asc:2.1
               VSMD.epd@asc:2.1 asc:2.1 -ignore EcucSws_1035
```

This example validates the VsMD file with the name *VSMD.epd* against the StMD file with the name *StMD.bmd*. Both files are regarded as Autosar 2.1 files. The ruleset used for the checks is *asc:21*, but the rule *EcucSws_1035*. is not checked.

4.12.4.4. Listing all rules used for VSMD checking

4.12.4.4.1. Syntax

To list all rules that are used for the `legacy vsmdcheck` command, use the `legacy listrules` command:

```
tresos_cmd.bat [<system_property>...] [-data <workspace>] legacy listrules
```

4.12.4.4.2. System properties

This command does only recognize the system properties common to all commands (see Section 4.12.1.3: System properties common to all commands link).

4.12.4.4.3. Example

The following example lists all rules used for VSMD checking with the `legacy listrules` command:

```
tresos_cmd.bat legacy listrules
```

4.12.4.5. Encrypting and singing modules

If you are developing an own module for use in EB tresos Studio as an EB partner, you can use the `crypto` command can be used to encrypt and sign modules. Please continue reading at Section 5.6: Signing modules with the crypto command for more information about this command.

4.12.5. Commands for converting and merging of files

This section shows you how to convert files to different formats and how to merge the content of files.

4.12.5.1. Merging files and converting between the XDM and Autosar formats

To merge and convert files, use the `convert` command. This command reads in one or more files with any known content type. It merges the content of these files and writes all read data to an output file with any specified content type.

4.12.5.1.1. Syntax

```
tresos_cmd.bat [<system_property>...] [-data <workspace>]
               legacy convert [<options>]
               <input-file>[@<type>]... <output-file>[@<type>]
```

4.12.5.1.2. System properties

In addition to the common system properties (see Section 4.12.1.3: System properties common to all commands), this command does also recognize the following system properties:

System Property	Description
`-DmergeConfigs=(true|false)`	If set to `true`, the parameters of the module configurations will be merged. This is useful, if you have split the parameters of a module configuration into several files (e.g. one file containing standard- and one containing vendor-specific parameters). If this property is set to `false`, no merge of the parameters is done. Instead, the content of the file read last overwrites the previous content. The default is `true` for this command.
`-DwriteDefaults=(true|false)`	Deprecated property. Can be used to force the `LOWER-MULTIPLICITY` and `UPPER-MULTIPLICITY` tags to be printed to the output-file even if their value is "1". The default value for this parameter is `false` and thus the tags are usually omitted. This property was used in pervious versions to achieve conformance to xsd-schemafiles what is now replaced by content types (see Section 4.2.2.3: File content types).
`-Duuids=(true|false)`	If set to `true`, new random unique IDs are created for each node. The default is `true`.
`-DsuppressReadOnlyWarnings=(true|false)`	If set to `true`, warnings regarding readonly configuration parameters are not logged or printed out. The default is `false` for this command.
`-DMapOptionalAsList=(true|false)`	This parameter is only usable if the content type of the output file is `xdm`. If set to `true`, a surrounding list for elements with LOWER-MULTIPLICITY= 0 and UPPER-MULTIPLICITY=1 is created. If set to `false`, no surrounding lists will be created. Instead, the affected elements be marked as optional. For backward-compatibility reasons, the default value is `true`.

System Property	Description
	Warning Leaving this property at the default, or explicitly setting it to `true` can lead to problems! This makes the XPaths of the affected elements depend on their LOWER-MULTIPLICITY value. If the LOWER-MULTIPLICITY is 0, a list is created. If the LOWER-MULTIPLICITY is later changed to 1, no list is created (with the UPPER-MULTIPLICITY always being at 1). The XPath is different in both cases. Code templates which then wish to address this parameter in the configuration need to be able to handle both cases. On the other hand, if you set this system property to `false`, the XPath stays the same in any case. **Tip** If you are writing a code-template, you can use the following XPath function in the template to determine if the parameter is configured or not: `node:exists(OptionalVariable)`.
`-DmergeAttributes=(true|false)`	This property is only useable, if the commandline contains `-DmergeConfigs=true`. It controls how to deal with the attributes of nodes that are present in more than one input file. The default value is `true`. If this property is set to `false`, the result will only contain the attributes and their values which were present when the node was read the first time. If this property is set to `true`, attributes with multiple values are merged so that the resulting attribute contains all values. How single-value attributes are dealt with, is controlled by the value of the property `-DoverwriteAttributes`.
`-DoverwriteAttributes=(true|false)`	This property is only useable, if the commandline contains `-DmergeAttributes=true`. It controls how to deal with the attributes of nodes that are present in more than one input file. The default value is `true`. If more than one input-file is used, this property controls how to handle the attributes of node present in several files. If the property is set to `true`, the values of the attributes are overwritten with their last occurrence. If the property is set to `false`, the value of the first occurrence of that attribute is used.

System Property	Description
`-DwriteXPathAttributes=(true|false)`	When writing an Autosar schema file, XPath attributes (e.g. a calculated DEFAULT or RANGE) will be ignored because they may access the configuration. In most cases, this would lead to unexpected results. To enforce executing XPath attributes and write their value, set this property to `true`. If the attribute anyhow does access the configuration it is suppressed. The default value is `false`.
`-DpreserveConfiguration=(true|false)`	With this property you may convert a pre-configuration file without changing the values configured. The default value is `false`; this value will add missing nodes and overwrite default values. To not add missing nodes and not overwrite values, set `-DpreserveConfiguration` to `true`. The imported configuration will not be changed in any way.

4.12.5.1.3. Options

The following options are supported by this command:

Parameter	Description
`-c <config-path>`	If this option is present, the module configuration files will be searched in the directory specified as `<config-path>`. By default, the directory `<base.path>\config` is used, with `base.path` being the value of the system property `-Dbase.path` (see Section 4.12.1.3: System properties common to all commands). If the system property is not set, an error is displayed.
`-o <output-path>`	If this option is present, the generated files will be written to the directory specified as `<output-path>`. By default, the directory of the target file is used.
`-n <config-name>`	Name of the module configurations of the config files to load. If this option is used, other module configurations contained in the specified files will be ignored. You can use this option multiple times.
`-u`	If this option is present, the generated files will contain Unix line endings.

4.12.5.1.4. Parameters

In addition to the common parameters `<system_property>` and `<workspace>`, this command does also recognize the following parameters:

Parameter	Description
`<input-file>[@<type>]`	Use `<input-file>` for the name of the file to read in. The parameter can be passed multiple times to load more than one file. The optional argument `@<type>` can be used to explicitly specify the content type of the file. See Section 4.2.2.3: File content types for more information on what content types are available.
`<output-file>[@<type>]`	Use `<output-file>` for the name of the file to write. The optional argument `@<type>` can be used to explicitly specify the content type of the file. See Section 4.2.2.3: File content types for more information on what content types are available.

4.12.5.1.5. Example

The following shows several examples of how to use the command `legacy convert`:

```
tresos_cmd.bat -DValidate=false -DMapOptionalAsList=false
               legacy convert in.epd out.xdm
```

The example above simply converts an epd-file to an xdm-file and marks optional parameters with the optional attribute instead of creating lists around them.

```
tresos_cmd.bat -DValidate=false -DMapOptionalAsList=false
               -Duuids=true legacy convert
               in1.epd in2.epd in3.xdm out.epc@asc:2.0
```

This example loads the three files *in1.epd*, *in2.epd* and *in3.xdm*. The content of the files is merged and saved to the output file called *out.epc*. The content of the output file conforms to Autosar 2.0. No validation is done during the conversion and new UUIDs are created for each node.

4.12.5.2. Using XDM files in different EB tresos Studio versions

If you are working with several releases of EB tresos Studio on the same configuration data, you may want to use an XDM created with a new version of EB tresos Studio in your old installation. Newer versions of EB tresos

Studio are always able to read old XDM files, but they may add new features to the XDM format that cannot be understood by older versions.

It is possible to use such a file with an older EB tresos Studio version. But you have to convert that file first into the older version. During that conversion, information that cannot be mapped into the old format may get lost!

Note

The XDM versions numbers are not related to the Autosar version numbers.

To convert a newer version of the XDM format to the older version, you can use the `legacy convert` command together with the EB tresos Studio installation which created that file.

The `legacy convert` command is explained in Section 4.12.5.1: Merging files and converting between the XDM and Autosar formats. All you need to do to convert between XDM file versions is to specify the correct content type for the files. The available content types are listed in Section 4.2.2.3: File content types. You can use any `xdm:*` content type.

4.12.5.2.1. Example

```
tresos_cmd.bat legacy convert new.xdm old.xdm@xdm:2.0
```

This example converts an XDM from a new EB tresos Studio version to the older XDM 2.0 format.

4.12.5.3. Upgrading the Autosar compliance of XDM files

The XDM files contain a representation of the Autosar data structures. Sometimes it is necessary to convert that representation to a newer Autosar revision because the newer revision contains incompatible changes. E.g. within Autosar 2.x choice conteiners do not have a SHORT-NAME, but in Autosar 3.0 they do. Instead of editing these changes manually inside the XDM file you can use the `legacy xdmconvert` command.

4.12.5.3.1. Syntax

```
tresos_cmd.bat [<system_property>...] [-data <workspace>]
               legacy xdmconvert <mode>
               <input-file>[@<type>]... <output-file>[@<type>]
               [-variant <variant>]
```

4.12.5.3.2. System properties

This command does only recognize the system properties common to all commands (see Section 4.12.1.3: System properties common to all commands link).

4.12.5.3.3. Parameters

In addition to the common parameters `<system_property>` and `<workspace>`, this command does also recognize the following parameters:

Parameter	Description
`<input-file>[@<type>]`	Use `<input-file>` for the name of the file to read in. The parameter can be passed multiple times to load more than one file. The optional argument `@<type>` can be used to explicitly specify the content type of the file. See Section 4.2.2.3: File content types for more information on what content types are available.
`<output-file>[@<type>]`	Use `<output-file>` for the name of the file to write. The optional argument `@<type>` can be used to explicitly specify the content type of the file. See Section 4.2.2.3: File content types for more information on what content types are available.
`<mode>`	Currently, there is only one mode supported: `asc2xToAsc30`. This mode converts an xdm file (representing an Autosar 2.x file) to an xdm file (representing an Autosar 3.0 file)
`-variant <variant>`	This parameter defines which implementation config variant shall be set at the MODULE-CONFIGURATION. Possible values are: ▶ `pre` for a pre-compile time configuration, ▶ `link` for a link-time configuration, and ▶ `post` for a post-build configuration. For further informations read the developers guide, Section 5.4: Upgrading an Autosar 2.1 Module to 3.0. This parameter is optional.

5 EB tresos Studio developer's guide

5.1. Introduction

This document describes how to create configurable modules for EB tresos Studio Studio. It is intended for developers of modules and should be read in addition to the User guide.

The document begins with a quickstart-chapter describing how to create a configurable module in EB tresos Studio followed by chapters describing the underlying DataModel, the storage-formats (xdm and Autosar), the XPath-language and the code-generator in a detailed way.

5.2. Interpretation of version information

EB tresos Studio is released on a halve year basis. Releases are named '2007.b', '2008.a'... Each release forms a separate variant to be able to supply bug fixes for different releases independently. To be able to supply short time bug fixes and updates during a release cycle EB submits multiple builds per release. Each build is identified by a suffix to the release version.

5.3. Setting up a new module

This chapter provides an overview on how to create a EB tresos Studio module. To maintain an overview, each topic in this chapter is described briefly. A more detailed description of each topic is contained in other chapters of this guide.

5.3.1. Prerequisites

You need either a standardized module definition (StMD) or a vendor specific module definition (VsMD).

EB tresos Studio contains a snapshot of the Autosar StMDs from each supported Autosar Release in the directories:

- `autosar/2.0`
- `autosar/2.1`
- `autosar/3.0`

To get single files for each module Change to the directory `autosar/3.0` of your EB tresos Studio installation and execute the following command:

```
create_epds_30.pl AUTOSAR_EcucParamDef.arxml
```

This creates several `.epd` files which contain the StMDs of the Autosar defined modules.

5.3.2. Creating a vendor specific module definition

According to the Autosar-specification, a vendor has to be derived from a VsMD from the corresponding Autosar StMD. For more details refer to the UsersGuide, chapter "Creating VendorSpecific module definitions (VSMD)".

To convert a StMD-file to a VsMD-file use the following command:

```
tresos_cmd.exe legacy vsmd <StMD> <VsMD> <package> [<ORIGIN>]
```

5.3.3. Converting files

To utilize special features provided by EB tresos Studio, parameter definition files should be shipped in XDM format. The XML format allows to define cross parameter checks, GUI annotation and to add other meta-information. To convert an Autosar file to XDM format or vice versa the `convert` command can be used:

```
tresos_cmd.exe legacy convert <infile>... <outfile>
```

The convert command can also be used to merge different files into one file by just providing multiple input files. E.g.:

```
tresos_cmd.exe legacy convert PduR.epd@asc:2.1 PduR.bmd@asc:2.1 PduR.xdm
```

5.3.4. Templates for new modules

EB tresos Studio ships with two templates for new modules that can be found in the directory `demos/`:

TemplateGenerator_TS_T00D0M0I0R0
: This template should be used when the template based code generator of EB tresos Studio should be used for generating code.

PublicApiTest_TS_T00D0M0I0R0
: This template should be used if a generator should be implemented in Java or other Java functionality should be used for the module.

The following sections describe how to setup a module with a template based code generator.

5.3.5. Setup of the file structure

Start by copying to your development environment and renaming the directory `TemplateGenerator_TS_T00D0M0I0R0`.

Replace the file `config/TemplateGenerator_TS_T00D0M0I0R0.xdm` by your own VsMD-file that was created as described above.

Remove all files and directories from within the generate-directory. This template-files can only be generated with the unchanged schematic-file of the template-plugin.

5.3.6. Adapting MANIFEST.MF

Change the following entries within `META-INF/MANIFEST.MF` according to your needs:

Bundle-Name
: Human-readable name for your plug-in.

Bundle-SymbolicName
: Unique name for your plug-in - a good practice is to use the directory-name of the plugin.

Bundle-Vendor
: The name of the vendor of the plugin

5.3.7. Adapting plugin.xml

Change the `plugin.xml` to your needs. The `plugin.xml` is an XML file used to register resources with EB tresos Studio. Registrations are grouped by so called *extensions*. For a first module setup adapting the three extensions defined in the demo module is sufficient.

5.3.7.1. dreisoft.tresos.launcher2.plugin.module

The extension `dreisoft.tresos.launcher2.plugin.module` registers a new module. The most important attribute in this extension is the attribute `id` beneath the `module` tag. This id is used in other extension in the form of an `moduleId` attribute to refer to this module.

The following attributes of the `extension` tag should be adapted:

id
: A unique id of your plug-in (this is NOT the moduleId).

name
: The human readable name of your plug-in.

The actual information of the extension can be found as attributes and sub-tags of the `module` module tag. Adapt the following attributes:

id
: The *module-id* for the module, this id is referenced by the extensionpoints `dreisoft.tresos.launcher2.plugin.configuration` and `dreisoft.tresos.launcher2.plugin.generator`.

label
: Provides a human readable name for the module. This is the module name that will be presented to the user in the GUI. If this value starts with a percent-sign, it must be written as key-value-pair (separated by a colon) within the file `plugin.properties`.

swVersion
: Provides the software version on which this module is based. See Section 5.3.7.4: Module versions for details.

specVersion
: Provides the version of the specification, this module is based. See Section 5.3.7.4: Module versions for details.

relVersion
: Provides the release version on which this module is based. See Section 5.3.7.4: Module versions for details.

categoryType
: Autosar type of the module: e.g. Can, CanIf, Lin, LinIf, Com, Os...

categoryLayer
: Layer of the module: should be one of "Service", "ECU Abs." or "MCAL"; used in the Configurator to group the configurations.

categoryCategory
: Category of the module: should be one of "IO", "Com", "Diag", "Mem" or "Sys"; used in the Configurator to group the configurations

categoryComponent
: Must be "ECUC"

With the `ecuType` sub-tag of the `module` module tag the ECU targets/derivates for which the module is valid can be restricted. If not ECU type is defined the module is valid for all targets. Multiple ecu types can be specified if the module is valid for multiple targets. The following attributes are known for the `ecuType`:

target
: The supported target of your module. The combination of target and derivative must be selected by the user when creating a new configuration-project if your module is to be used.

derivate (optional)
: The supported derivative of your module.

Here you can see a sample of the `module` extension:

```
<extension point="dreisoft.tresos.launcher2.plugin.module"
  id="YourID"
  name="Your plugins name">

  <module id="Your_Module_Id"
          label="Your modules human readable name"
          mandatory="false"
          legacy="false"
          allowMultiple="false"
          description="Your modules description"
          copyright="YourCopyright"

          swVersionPrefix="2007.b.pl1"
          swVersionMajor="2"
          swVersionMinor="0"
          swVersionPatch="1"
          swVersionSuffix=""

          specVersionPrefix=""
```

```
          specVersionMajor="1"
          specVersionMinor="9"
          specVersionPatch="0"
          specVersionSuffix="Rev.17"

          relVersionPrefix="AUTOSAR"
          relVersionMajor="2"
          relVersionMinor="1"
          relVersionPatch="0"
          relVersionSuffix=""

          categoryType="YourType"
          categoryLayer="Service|ECU Abs.|MCAL"
          categoryCategory="IO|Com|Diag|Mem|Sys"
          categoryComponent="ECUC">
    <ecuType target="YourTarget" derivate="YourDerivat"/>
  </module>
</extension>
```

5.3.7.2. dreisoft.tresos.launcher2.plugin.configuration

The extension `dreisoft.tresos.launcher2.plugin.configuration` registers configuration specific resources for the module like the parameter definition, the provided editors, pre- and recommended configuration.

The attributes of the extension tag are standard attributes already described for the module tag.

The toplevel tag beneath the extension tag is called `configuration`. This tag makes the association the module defined by a `module` extension via the attribute `moduleId`. The value of this attribute must match the `id` of the `module` tag of a previous module registration.

Configuration information registered via the configuration extension is grouped into various sub-tags of the `configuration` tag.

5.3.7.2.1. Parameter definition, pre-, recommended configuration

Via the sub-tag `schema` the parameter definition and the pre- and recommended configuration can be registered.

The various information registered by the `schema` tag is grouped into sub-tags.

One mandatory sub tag named `manager` names the Java class that loads a schema resources. The class is selected via the attribute `class` which must be set to `dreisoft.tresos.autosar2.resourcehandling.AutosarSchemaManager`.

Schema information is loaded from files within the module plug-in. The files to load are registered with the tag `resource`. Multiple files can be registered. The resource tag accepts the following attributes:

id (optional)
: The `id` is optional. If pre- or recommended configuration should be supplied this attribute can be used to reference to the file from which to load the pre- or recommended configuration.

value
: The `value` attribute contains a path relative to the root directory of the module plug-in that says where to find the XDM file from which to the load the schema information.

type
: The type of the resource must be set to `XDM`.

EB tresos Studio supports multiple pre- configurations per module which are registered via the `plugin.xml`. If at least one pre- configuration is specified the user must select one pre- configuration when creating a module configuration for the module. If multiple pre- configurations are specfied the user is free to select one of the provided.

A pre- configuration is described via the sub-tag `preconfig` of the `schema` tag. The following attributes are available:

name
: The `name` attribute provides the name of the pre- configuration that is shown to the user.

resourceId
: The `resourceId` attribute selects the resource file from which to load the pre- configuration. The supplied id must match an id of a `resource` tag supplied for this module.

path
: The `path` attributes provides an XPath or ASPath to the module configuration in the referenced XDM that forms the pre- configuration.

description (optional)
: The `description` attributes provides a supplementary text that is shown to the user when selecting the pre- configuration in the GUI.

default (optional)
: The `default` attributes selects the pre- configuration that is presented to the user a initially selected pre-configuration when adding a new module configuration.

EB tresos Studio supports multiple recommended configurations per module which are registered via the `plugin.xml`. If multiple pre- configurations are specfied the user is free to select one of the provided. The user also is allowed to not load any recommended configuration.

A recommended configuration is described via the sub-tag `recconfig` of the `schema` tag. The following attributes are available:

name
: The `name` attribute provides the name of the recommended configuration that is shown to the user.

resourceId
: The `resourceId` attribute selects the resource file from which to load the recommended configuration. The supplied id must match an id of a `resource` tag supplied for this module.

path
: The `path` attributes provides an XPath or ASPath to the module configuration in the referenced XDM that forms the recommended configuration.

description (optional)
: The `description` attributes provides a supplementary text that is shown to the user when selecting the recommended configuration in the GUI.

default (optional)
: The `default` attributes selects the recommended configuration that is presented to the user a initially selected recommended configuration when adding a new module configuration. If none of the recommended configuration is the default one no recommended configuration is loaded by default.

5.3.7.2.2. Module configuration

How the module configuration of a new instance of the module is created is defined by the `data` tag.

Via the tag `manager` the Java class that manages the module configurtion can be selected. The attribute `class` must be set to `dreisoft.tresos.autosar2.resourcehandling.AutosarConfigManager`.

The tag `schemaNode` selects the module definition that should be used to create the module configuration. The attribute `path` must contains an XPath or ASPath pointing to a module definition in one of the registered schema files.

5.3.7.2.3. Registering editors

The tag `editor` allows to register editors that are presented to the user for editing the module configuration. EB tresos Studio provides an implementation of a generic configuration editior that can be registered via this tag. An editor registration for a generic configuration editor has the following form:

```
<editor id="AnyUniqueId"
        label="AnyLabel"
        tooltip="AnyTooltip">
<class
class="dreisoft.tresos.launcher2.editor.GenericConfigEditor"
>
```

```
<parameter name="schema" value="ASPath to your MODULE-DEF"/>
</class>
</editor>
```

The following attributes must be edited module specific:

id
: An id that must be unique to identify the editor.

label
: A label for the editor shown in the GUI.

tooltip
: A tooltip for the editor shown in the GUI.

The generic configuration editor required the following parameters to be set via the `parameter` tag:

schema
: Path to the module definition used to create the module configuration. The value must match the path given in the `data` tag.

5.3.7.2.4. Code sample

The following sample shows a complete configuration extension:

```
<extension point="dreisoft.tresos.launcher2.plugin.configuration"
id="AnyUniqueId"
name="AnyName">
<configuration moduleId="YourModuleId">
<schema>
<manager
class="dreisoft.tresos.autosar2.resourcehandling.AutosarSchemaManager"
/>
<resource value="RelativePathToYourVsMD" type="xdm"/>
</schema>

<data>
<manager
class="dreisoft.tresos.autosar2.resourcehandling.AutosarConfigManager"
 />
<schemaNode path="ASPath to your MODULE-DEF"/>
```

```
</data>

<editor id="AnyUniqueId"
label="AnyLabel"
 tooltip="AnyTooltip">
<class
class="dreisoft.tresos.launcher2.editor.GenericConfigEditor"
>
<parameter name="schema" value="ASPath to your MODULE-DEF"/>
<parameter name="title" value="AnyTitle"/>
</class>
</editor>
</configuration>
</extension>
```

5.3.7.3. dreisoft.tresos.launcher2.plugin.generator

Via the generator extension a code generator can be registered for the module. A module can have any number of code generators. The generator extension supplies the tag `generator` as its single tag. The `generator` tag supports the following attributes:

id
: A generator can be given a unique id with this attribute.

moduleId
: The id of the module extension used to register the module.

class
: The Java class that implements the code generator.

step (optional)
: The attribute `step` allows to select the phase of the code generation in which the generator is called. Code generation runs in three phases or steps: pre, default, post that are run in this order. Via the values `pre` and `post` the generator can be scheduled into the pre or into the post step.

The template based code generator is implemented in the Java class `dreisoft.tresos.launcher2.generator.TemplateBasedCodeGenerator`. Via the tag `parameter` the template based code generator can be customized. The following parameters can be used:

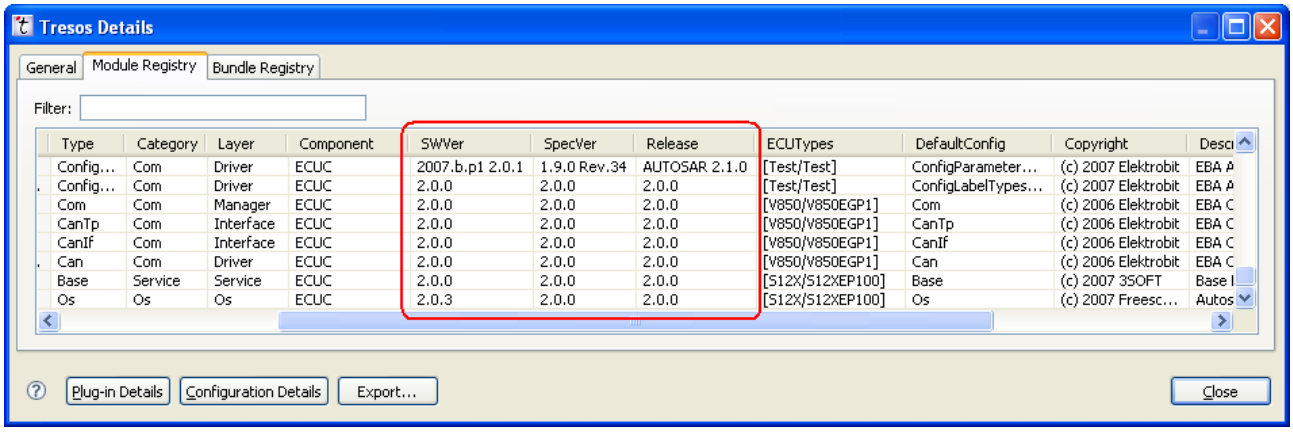
templates
: Selects a path relative to the module plug-in root directory benath which the code templates can be found.

```
<extension point="dreisoft.tresos.launcher2.plugin.generator"
           id="StandardModule_TS16D4M2I0R0GenExt"
           name="Module PlugIn TemplateGenerator">

  <generator
    class="dreisoft.tresos.launcher2.generator.TemplateBasedCodeGenerator"
    id="AnyUniqueId"
    moduleId="YourModuleId">
    <parameter name="templates" value="RelativeTemplatePath"/>
  </generator>
</extension>
```

5.3.7.4. Module versions

Within the plugin.xml, the software version, specification version and release version can be specified for each module. This informations are shown to the user in the "Tresos Details" dialog which shows all installed modules and can be opened using the menu "Help/Tresos Studio Details..." and there on the tab page "Module Registry".



Type	Category	Layer	Component	SWVer	SpecVer	Release	ECUTypes	DefaultConfig	Copyright	Desc
Config...	Com	Driver	ECUC	2007.b.p1 2.0.1	1.9.0 Rev.34	AUTOSAR 2.1.0	[Test/Test]	ConfigParameter...	(c) 2007 Elektrobit	EBA A
Config...	Com	Driver	ECUC	2.0.0	2.0.0	2.0.0	[Test/Test]	ConfigLabelTypes...	(c) 2007 Elektrobit	EBA A
Com	Com	Manager	ECUC	2.0.0	2.0.0	2.0.0	[V850/V850EGP1]	Com	(c) 2006 Elektrobit	EBA C
CanTp	Com	Interface	ECUC	2.0.0	2.0.0	2.0.0	[V850/V850EGP1]	CanTp	(c) 2006 Elektrobit	EBA C
CanIf	Com	Interface	ECUC	2.0.0	2.0.0	2.0.0	[V850/V850EGP1]	CanIf	(c) 2006 Elektrobit	EBA C
Can	Com	Driver	ECUC	2.0.0	2.0.0	2.0.0	[V850/V850EGP1]	Can	(c) 2006 Elektrobit	EBA C
Base	Service	Service	ECUC	2.0.0	2.0.0	2.0.0	[S12X/S12XEP100]	Base	(c) 2007 3SOFT	Base I
Os	Os	Os	ECUC	2.0.3	2.0.0	2.0.0	[S12X/S12XEP100]	Os	(c) 2007 Freesc...	Autos

Figure 5.1. Tresos details

Beside this, the user can see the version informations for each module in its properties. It can be opened by opening the properties of a project, selecting a module within the "Modules" section and clicking on the "Properties" button.

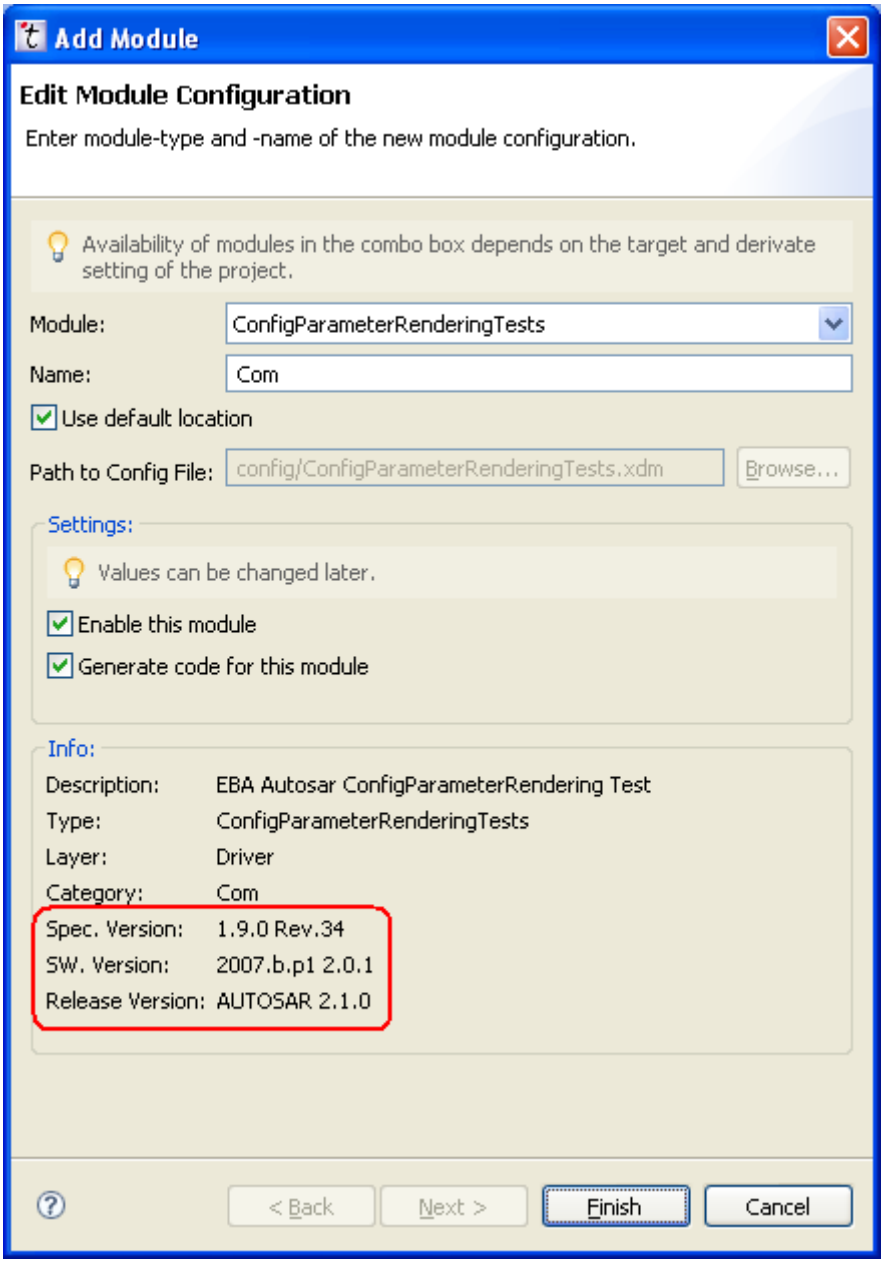


Figure 5.2. Module properties

Each of the versions may optionally have a prefix and/or a suffix. This are only informal informations and not evaluated in any way. But they are shown to the user and thus shall be self-explaining.

5.3.7.4.1. Software version

Provides the software version on which this module is based on. This might be e.g. software version of the contained code-generator or of the software which handles the generated code. This might also be used to define module-dependencies in future releases.

5.3.7.4.2. Specification version

Provides the version of the specification, this module is based on. This might be used to define module-dependencies in future releases.

5.3.7.4.3. Release version

Provides the release version on which this module is based on. The user has to define a "Release Version" for each newly created project which restricts the available modules for that project. For this, only the major and minor version is used.

5.3.8. Code templates

If not changed in the `plugin.xml` the code generator searches the code templates within the plug-in underneath the directory `generate`.

A good practice is to create a directory within the generation-path with the name of your plugin since there might be several modules having the same output-path.

5.3.9. Adding the module to EB tresos Studio

To install the plug-in to EB tresos Studio, copy the whole directory to the EB tresos Studio installation into the directory `plugins`.

Due to a bug in Eclipse, all directories named `org.eclipse.*` underneath the `configuration` directory of the EB tresos Studio installation have to be removed.

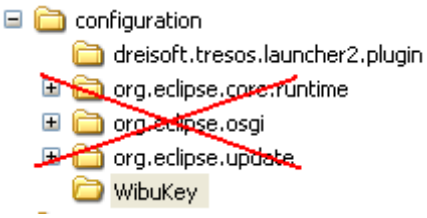


Figure 5.3. Directories to delete

The `org.eclipse.*` directories contain cached information about installed plugins. Not deleting them can cause several undefined effects.

After restarting EB tresos Studio the new plug-in will be available.

If you have purchased EB tresos AutoCore together with EB tresos Studio, you can also integrate the plugin-in with the User Build environment. You will find the instructions on how to do this in the EB tresos AutoCore

documentation: `EB tresos AutoCore documentation/EB tresos AutoCore user's guide/Build environment`.

5.4. Upgrading an Autosar 2.1 Module to 3.0

Converting an Autosar 2.1 module into an Autosar 3.0 must be done with the following order:

1. Use the "legacy vsmd" commandline command to create a VSMD XDM file from the Autosar 3.0 StMD file.
2. Merge all vendor specific changes from the existing Autosar 2.1 parameter definition into the newly created Autosar 3.0 vendor specific parameter definition file.
3. Use the "legacy vsmdcheck" commandline command to ensure that the new parameter definition conforms to Autosar 3.0.

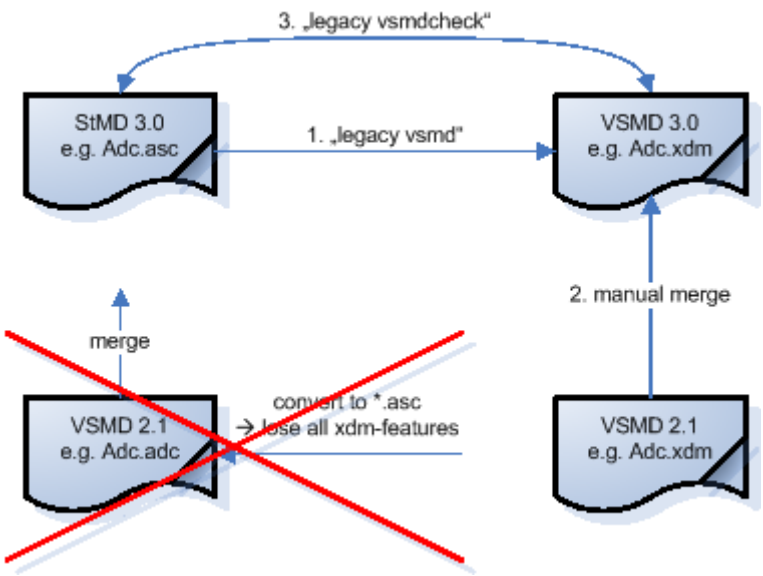


Figure 5.4. Tresos details

Note

Do not convert existing XDM-file to the Autosar format as all XDM specific features would be lost in this way.

In the following sub sections the changes between Autosar 2.1 and Autosar 3.0 which must be considered when switching a parameter definition to Autosar 3.0 are presented. For further information please refer to the Autosar documents.

5.4.1. Choices

Since Autosar 3.0 choice within XDM files are of type `IDENTIFIABLE` (in previous Autosar they had been of type `CHOICE`):

wrong:

```
<v:chc name="CanTrcvAccess" type="CHOICE">
```

correct:

```
<v:chc name="CanTrcvAccess" type="IDENTIFIABLE">
```

This is especially important when merging the old and the new vsmd XDM file. The VSMD checker will marked invalid `v:chc` tags as an error.

Note

As choices are now represented as `IDENTIFIABLE` elements in the ECU Configuration Description all Autosar paths to elements beneath a choice in the configuration must be adjusted. This means that existing configuration cannot be used without change even if the schema itself does not change between Autosar 2.1 and Autosar 3.0. The same holds for pre configurations and recommended configurations.

5.4.2. Configuration variant

With Autosar 3.0 configuration variants have been introduced. With this new features one module parameter definition can be used to describe a compile time, and a link time and a post build module implementation. This allows the user to select the actual variant of the module when editing the module configuration. Which configuration variants a module actually supports is described in the VSMD. So the module developer is in charge to decide which variants the module will support.

The Autosar StMD normally defines all variants for each module. The creator of a VSMD module definition has to decide which variants are actually supported and remove the excessive variants from the VSMD file. To do this the variable `IMPLEMENTATION_CONFIG_VARIANT` has to be adjusted:

```
<v:var name="IMPLEMENTATION_CONFIG_VARIANT" type="ENUMERATION">
  <a:a name="LABEL" value="Config Variant"/>
  <a:da name="DEFAULT" value="VariantPreCompile"/>
```

```
  <a:da name="RANGE">
     <!-- Remove not supported variants from this list -->
     <a:v>VariantPreCompile</a:v>
     <a:v>VariantLink</a:v>
     <a:v>VariantPostBuild</a:v>
  </a:da>
</v:var>
```

Each configuration parameter specifies its configuration class in each configuration variant. For all parameters taken over from the StMD these lists have to be adjusted. For all vendor specific parameters a corresponding mapping has to be added. The configuration class mapping looks as follows in the XDM:

```
<v:var name="AdcHwUnitId" type="INTEGER">
  <a:a name="IMPLEMENTATIONCONFIGCLASS"
       type="IMPLEMENTATIONCONFIGCLASS">
    <!-- adjust list of variant mappings -->
    <icc:v class="PreCompile">VariantPreCompile</icc:v>
    <icc:v class="Link">VariantPostBuild</icc:v>
    <icc:v class="PostBuild">VariantPostBuild</icc:v>
  </a:a>
</v:var>
```

5.4.3. ADMIN-DATA

Since Autosar 3.0 `ADMIN-DATA` must be added beneath `MODULE-DEF` like this:

```
<d:chc name="CanIf" type="AR-ELEMENT" value="MODULE-DEF">
  <v:ctr type="MODULE-DEF">
    <a:a name="ADMIN-DATA" type="ADMIN-DATA">
      <ad:ADMIN-DATA>
        <ad:LANGUAGE>EN</ad:LANGUAGE>
        <ad:DOC-REVISIONS>
          <ad:DOC-REVISION>
            <ad:REVISION-LABEL>1.2</ad:REVISION-LABEL>
            <ad:ISSUED-BY>anonymous</ad:ISSUED-BY>
            <ad:DATE>2008-5-26T21:32:52</ad:DATE>
            <ad:MODIFICATIONS>
              <ad:MODIFICATION>
```

```
            <ad:CHANGE>
              <ad:L-2 L="DE">ChangeTitle</ad:L-2>
            </ad:CHANGE>
            <ad:REASON>
              <ad:L-2 L="EN">Reason</ad:L-2>
            </ad:REASON>
          </ad:MODIFICATION>
        </ad:MODIFICATIONS>
      </ad:DOC-REVISION>
    </ad:DOC-REVISIONS>
  </ad:ADMIN-DATA>
</a:a>
```

5.5. Module Transformer

Module Transformers are used to convert an existing module configuration to another version or target/derivate. If you have an AUTOSAR 2.1 configuration project and want to import it to an AUTOSAR 3.0 project, the module specific transformers are called to convert the configuration to the newer version.

5.5.1. Registering a Module Transformer

To register a module transformer, the `dreisoft.tresos.launcher2.api.plugin.moduletransformer` extension point has to be registered in the new module's plugin.xml. Conduct the following steps:

Add the extension point via the plugin.xml editor.

Enter the module id of the module to which it belongs and assign an id for the transformer. The module id is MyModule_TS_T16D4M2I1R0 then the id for the transformer should be something like MyModuleTransformer_TS_T16D4M2I1R0. The moduleId specifies the target module to which an old configuration shall be transformed. The version of the source module can be specified within the `enablement` attribute which is explained in one of the next paragraphs.

Add subelement to the moduletransformer named `type`. The type how good the tranformer is (fully, partially, fallback). "fully" wins against a "partially" and "partially" wins against "fallback".

Add subelement to the moduletransformer named `transformer`. This element defines the class, that will do the transformation. The class must implement dreisoft.tresos.launcher2.module.transform.IModuleTransformer. To support the full functionality of the extension-point it must implement dreisoft.tresos.launcher2.module.transform.AbstractModuleTransformer. Clicking on the *class:* link will open the class creation wizard, with the right abstract class already added.

Add subelement to the moduletransformer named `transformation`. Enter the kind of transformation that will be performed. Possible conversions are "version" or "target".

Add subelement to the moduletransformer named `enablement`. Within the enablement you can enter expression, that can query some variables of a module and compare them to values you have entered. If the expressions evaluate to true, the transformer will be enabled for this module. The following variables can be used:

- id: Module-Id (e.g. PduR_TS_T16D4M2I0R0)
- type: Type of the module (e.g. PduR)
- category: The module category (e.g. Com)
- layer: The module layer (e.g. Service)
- relVersion: The release version (e.g. 2.1, 2.*, 2.1.0,2.1.*)
- specVersion The module spec version (e.g. 2.0.1, 2.*, 2.1)
- swVersion: The software version of the module (e.g. 2.1.1,2.*,2.1)
- ecuTypes: The target/derviates supported (e.g. V850/V850EGP1, V850, S12X)

5.5.1.1. Examples

Expression Examples:

```
<enablement>
<and>
<with variable="ecuTypes"><iterate value="V850"/></with>
<with variable="relVersion"><equals value="2.0.*"/></with>
</and>
</enablement>

<enablement>
<or>
<with variable="ecuTypes"><iterate value="V850"/></with>
<with variable="ecuTypes"><iterate value="S12X"/></with>
</or>
</enablement>

<enablement>
<and|or|not>
<with variable="ecuTypes">
<iterate value="TRICORE/TC1766"/>
</with>
```

```
</and|or|not>
</enablement>

<enablement>
<with variable="type"><equals value="Can"/></with>
</enablement>
```

The following example registers a module transformer, that transforms a module with name MyModule from version 2.0 to 2.1.

```
<extension id="MyModuleTransformerExtId"
           point="dreisoft.tresos.launcher2.api.plugin.moduletransformer">
    <moduletransformer id="MyModuleTransformer_TS_T16D4M2I1R0"
                       moduleId="MyModule_TS_T16D4M2I1R0">
      <type value="partially"/>
      <transformation type="version"/>
      <enablement>
        <and>
        <with variable="type"><equals value="MyModule"/></with>
        <with variable="relVersion"><equals value="2.0"/></with>
        </and>
      </enablement>
      <transformer class="eb.autocore.mymodule.transformer.Rel20Transformer"/>
    </moduletransformer>
</extension>
```

5.5.2. Implementing a Module Transformer Class

If you followed the steps in the previous section, you should have a java class that extends the AbstractModuleTransformer. The only method you have to implement is the abstract method

```
public boolean doTransform( ModuleDescription sourceModule,
                            ModuleDescription targetModule,
                            DCtxt moduleConfiguration )
```

The parameters will give you access on information about the source and the target modules, as versions, target, etc. The DCtxt contains the source configuration of the module, that shall be transformed. The configuration is not bound to a schema file, so you can delete complete nodes as containers, variables, references from the config and add them at another location. The DCtxt gives you access to methods that allow you to conduct those operations. The methods to operate on an unbound configuration all start with the prefix "ns" (e.g. nsAddContainer).

5.5.3. Documentation

If you installed the eclipse tools from the EB tresos Studio tools folder, the documentation of the DCtxt (datamodel Public API) can be found in the "Eclipse Help Contents" with the title "Public API Java Documentation". If you have EB tresos Studio set as target platform, the help is also available as javadoc in the java eclipse editor.

5.5.4. Chaining of Module Transformers

Module transformers can also be chained. That means, that there must not be one transformer for each possible combination of version-to-version conversion. If you already have a transformer that transforms from version 2.0 to 2.1 and now you want to transform your module configuration from version 2.0 to 3.0 you only need to implement a transformer that transforms from version 2.1 to 3.0. At last you need to register a *fully* transformer, that just calls the right transformers to convert from 2.0 to 3.0.

5.6. Signing modules with the crypto command

This chapter describes how to sign a module.

5.6.1. Background

Partners of EB receive a special redistributable license that allows them to redistribute their modules on their own. To do this these modules must be signed with a special cryptographic key. Only modules signed with this key can be redistributed with this special license.

In the future EB may also provide the possibility for partners to encrypt parts of their modules but currently this feature has not been made available for outside EB.

5.6.2. Prerequisites

Partners of EB must use the crypto tool to sign their modules. To be able to operate the tool partners receive:

1. a redistributable license file (.lic): This is the undongled license which can be delivered together with an installation of.
2. crypto license file (.lic): Dongled license for the crypto tool.
3. signature key (.dat): This key file is needed by the crypto tool to sign modules.

5.6.3. Crypto command

Modules are signed by using the crypto command via the commandline. Before the command can be used the crypto license file must have been installed either via the GUI (`Help->Licenses`) or by copying the file to the `bin/` directory.

Assuming the module is installed in the directory `plugins/Can_Vendor` with the following file structure:

```
config/Can_Vendor.xdm
generate/Can_Cfg.c
```

In this case the module can be signed with:

```
$ tresos_cmd.bat crypto -v \
                        -skf tresos_signingkey_Vendor.dat \
                        -ski Can_Vendor \
                        -files \
                        config/Can_Vendor.xdm;generate/Can_Cfg.c \
                        plugins/Can_Vendor
```

where:

- tresos_signingkey_Vendor.dat: is the signature key file.
- Can_Vendor (-ski option): is the provider-id of the module to be signed. The provider-id can be found within your license-file (`*.lic`) in field CUSTOMER.

The tool signs all files given with the `-files` commandline option. Files are separated by ';'. The paths are relative to the module plug-in directory given as last parameter. The tool creates a file `META-INF/CRYPTO-MANIFEST.MF` that contains the created signatures.

The following files must be signed:

- Parameter definition files (.xdm)
- All code templates and macro files (generally speaking all files that are processed by the code generator.

5.7. The DataModel

The DataModel provides a very flexible in-memory representation of configuration data.

All nodes are stored within a tree-structure. Thus each node has one parent and as many child-nodes as desired. Only the top-level-node of the tree does not have a parent.

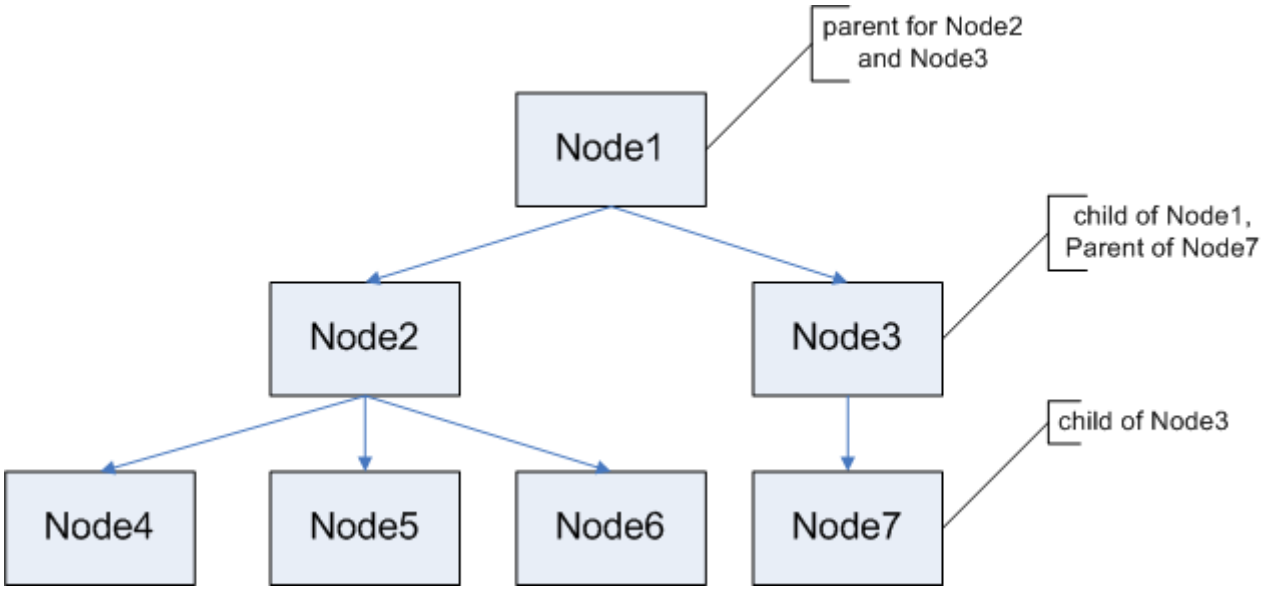


Figure 5.5. Simple Node-Tree

Configuration data is stored in a data-tree. The structure of this tree is described by a schematic-tree. The schema-tree and the data-tree always have the same structure which means that:

- Every data-node has a single schema-node which describes it.
- Each data-node has a reference to the schematic-node that describes it.
- The parent of a data-node refers to the parent of the corresponding schema-node. There never is a leap node between them. This ensures the similarity of the schematic- and data-tree-structure.

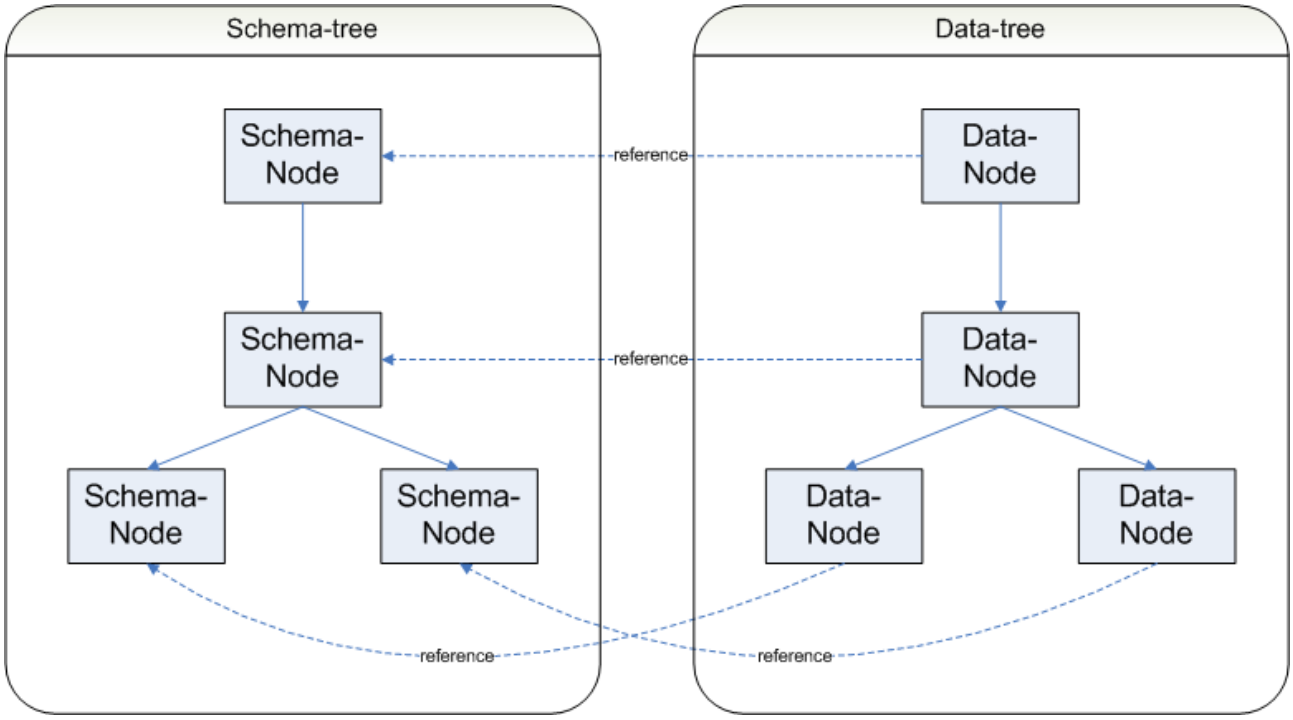


Figure 5.6. Simple schematic- and data-tree

The schematic- and data-trees are represented by different node types:

Schema	Data	Description
container-node	container-node	Stores a fixed list of nodes. The schematic-node contains the same amount of children as the data-node
list-node	list-node	Stores a variable list of nodes. The schematic-node always contains only a single node. The data-node contains one node for each configured (added) value.
choice-node	choice-node	Stores a fixed list of nodes. The schematic-node contains all available choices as children. The data-node contains the selected node as a child.
boolean-node integer-node string-node float-node	variable-node	The schema-node describes the data-type and range of the data-node. The data-node stores one configuration entry. These nodes cannot contain other nodes.
reference-node	reference-node	The data-node refers to another node. These nodes cannot contain other nodes.

Table 5.1. Schematic-Nodes and Data-Nodes

5.7.1. Schematic-tree

The schematic-tree itself does not store any configuration data. It just describes the data-tree and provides meta-information about:

- types and ranges of configuration data variables
- default-values for configuration data variables
- layout information for the visual component
- tool-tip
- online help
- etc.

Values of data-nodes are stored as strings without further type-information. Thus, the value of a data-node may only be interpreted correctly with the meta-information of the schematic-node referred.

5.7.1.1. Example

The four data-nodes in the following diagram hold the same string-value. With support of the schematic-node, the values get their data-type: integer, float, string and an invalid integer (attributes like RANGE will be explained in Section 5.8: The XDM format).

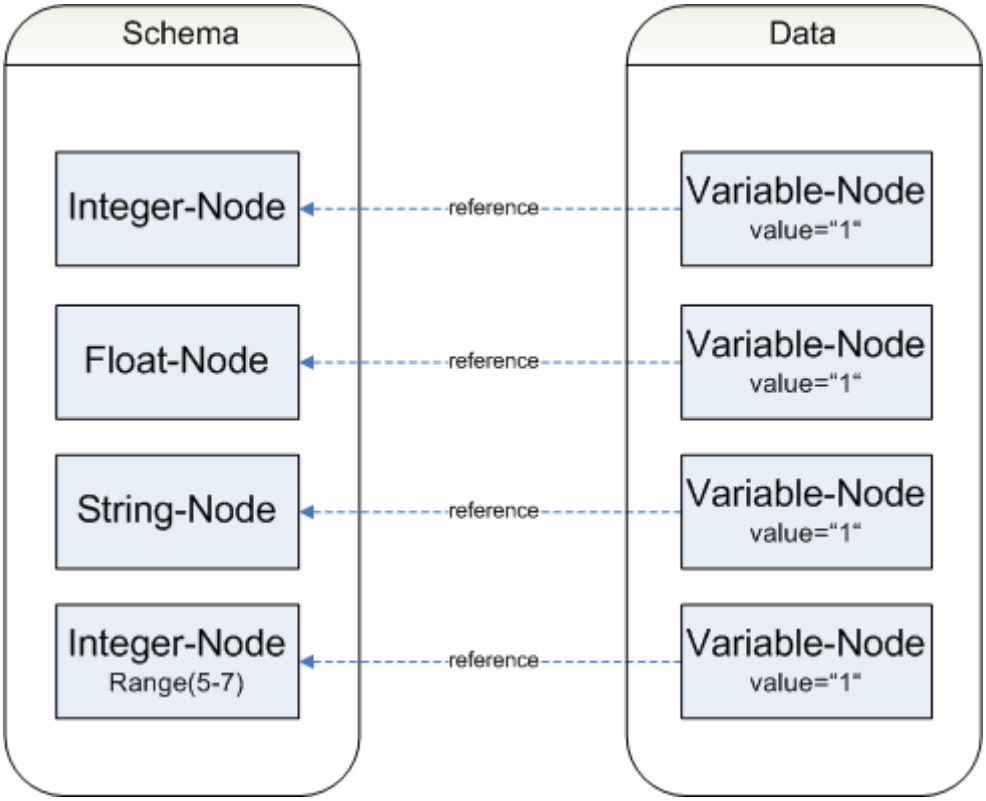


Figure 5.7. Nodes in Schema and Data

5.7.1.2. Features of Schema-Nodes

A schema-node permits:

- automatically instantiating corresponding configuration data-nodes
- filling in default values
- verifying entered data
- rendering dialogs which allow the user to fill in the configuration.

Multiple data-trees may refer to the same schema-tree, since the schema-tree only holds meta-information which is unique for all data-trees.

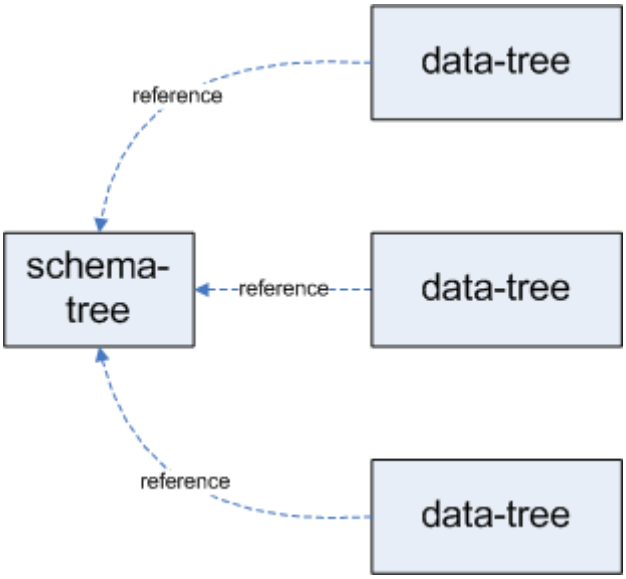


Figure 5.8. Multiple data-trees referring to one schematic-tree

5.7.2. Data-Tree

A data-node stores the configured value(s). For each node within the schema-tree, corresponding data-nodes can be created within the data-tree.

5.7.3. Node-Attributes

Each node can have several attributes. Each attribute may store multiple string values and is refered to by name.

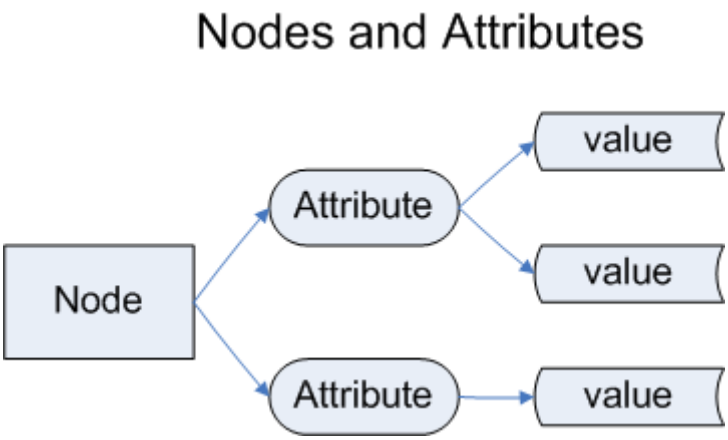


Figure 5.9. Nodes and attributes

For example, the DEFAULT-attribute holds the default-value for a data-node. The RANGE-attribute may contain several values that define ranges for a data-node.

When schema or configuration data is loaded from XDM-files, attributes are explicitly set as they are defined in the XDM-file. When an Autosar configuration format is loaded, attributes are set implicitly (e.g. the RANGE-attribute will be calculated from ENUM-LITERAL-DEF).

The DataModel recognizes two types of attributes:

- Passive attributes. The values are set explicitly.
- Automatic attributes. Their values will be calculated each time the data in the tree changes as their value depends on the current configuration. Their values cannot be set explicitly.

5.7.4. List-Representation within the DataModel

A particular feature of a list-node is that there may be more children within the data than in the schema. Within the schema-tree, a list-node always has exactly one child. This child describes the children of the corresponding list-node within the data-tree. All list-children within the data-tree refer to the single list-child within the schema-tree.

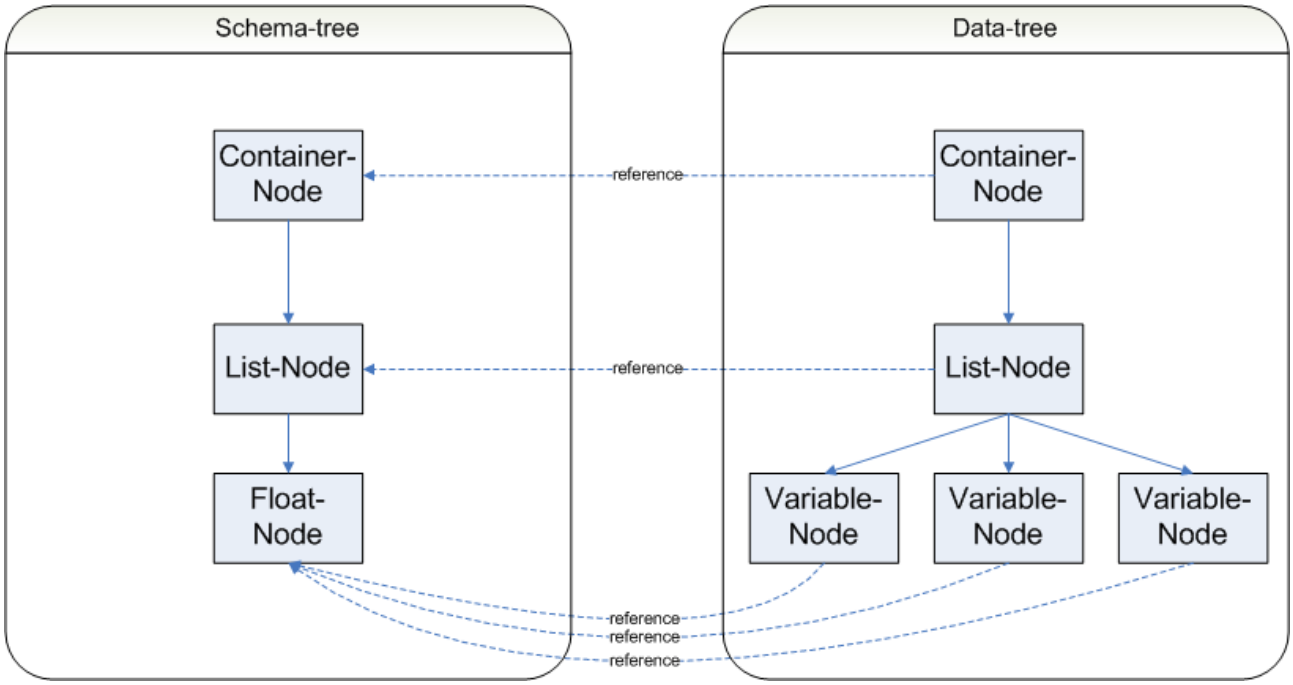


Figure 5.10. List-Representation

5.7.5. Choice-representation within the DataModel

A choice provides the possibility to choose a single node from of a list of sub-nodes. Within the schema-tree all available nodes are children of the choice-node. Within the data-tree the choice contains a child that represents the node which the user selected. The choice-node within the data provides the name of the chosen node as the node-value.

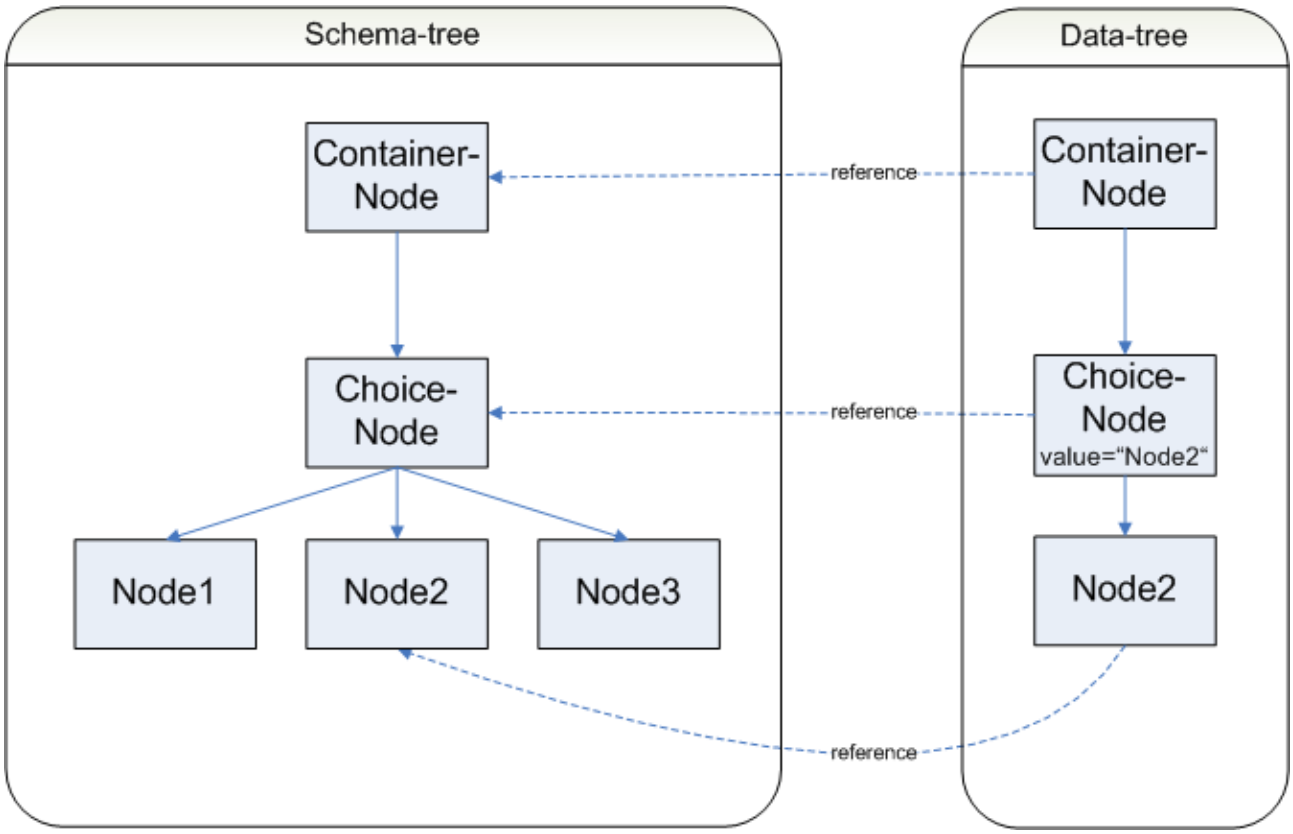


Figure 5.11. Choice-Representation

5.8. The XDM format

The native persistency format for storing a DataModel is the XDM-format. The XDM-format is an XML data-format that allows storing the node structure including all data (attributes, node names) of a DataModel on disk. The XDM format was also used in the first version of the DataModel. The XDM format presented in this chapter therefore is XDM version 3.0 as it was drastically changed to conform to the new node structure which is needed for Autosar.

5.8.1. Concepts

The XDM format stores:

- Nodes (schema-nodes, data-nodes...)
- Node Attributes

To keep the format extensible, multiple XML-Namespaces are used to differentiate elements of the DataModel.

The DataModel supports different kinds of nodes (for which different name spaces are defined):

generic nodes:
: Nodes that are neither schema-nodes nor data-nodes.

schema-nodes:
: Schema-nodes define configuration parameters and the structure of configuration parameters. They define type, structure and attribute defaults for data-nodes.

data-nodes:
: Data-nodes represent configuration parameters. A data node is *bound* to a schema-node that describes valid parameter ranges for the data-node.

5.8.2. Top level structure

An XDM-file is enclosed by the *datamodel* XML-tag:

```
'<datamodel'
   'version="3.0"'
   [ 'xmlns="http://www.tresos.de/_projects/DataModel2/06/root.xsd"' ]
   [ 'xmlns:a="http://www.tresos.de/_projects/DataModel2/06/attribute.xsd"' ]
   [ 'xmlns:v="http://www.tresos.de/_projects/DataModel2/06/schema.xsd"' ]
   [ 'xmlns:d="http://www.tresos.de/_projects/DataModel2/06/data.xsd"' ]
   '>'

...

'</datamodel>'
```

Attribute	Description
version	Must be set to `3.0`
xmlns	Declares the XML-namespaces for the tags of the DataModel. The namespace for generic node-tags and the datamodel-tag is: `http://www.tresos.de/_projects/DataModel2/06/root.xsd`
factory	The name of the factory to be used to create nodes.

Table 5.2. XML-Attributes

Child-tags:

- All node-tags

The datamodel-tag is not represented by its own node in the datamodel.

5.8.3. Nodes

Node tags are used to create nodes in the datamodel. There are various node tags available, one for each kind of node (container, list, variable). The XML-namespace is used to group nodes to categories (data-nodes, schema-nodes...). All node tags have a common form:

```
'<' namespace-prefix ':' core-node-type [ 'name="' node-name '"' ]
                                        [ 'type="' type-string '"' ]
                                        [ 'value="' value '"' ]
                                        [ 'factory="' factory-name '"'
                                          [xmlns:...]
                                        ]
                                        '>'

'</' namespace-prefix ':' core-node-type '>'
```

The possible namespace-prefixes (e.g. 'd', 'v', 'a') are described in the chapters Section 5.8.4: Attributes, Section 5.8.5: Generic nodes, Section 5.8.6: Schema-nodes and Section 5.8.7: Data-nodes.

See chapter Section 5.8.8: Factories for details about the factory and xmlns attributes.

Attribute	Description
name	The name of the node. Any node can have a name. Some nodes require that their child nodes have a unique name.
type	The optional type attribute in addition to the node-tag determines the node type (the class that gets created for the node, allowed child nodes, attributes...).
value	Each node can have a value which is set with this attribute.
factory	The name of the factory to be used to create nodes.

Table 5.3. XML-Attributes

Child-tags:

- All node-tags
- All attribute-tags

5.8.4. Attributes

Feature	Description
Namespace	http://www.tresos.de/_projects/DataModel2/06/attribute.xsd
Default Prefix	a

Table 5.4. Namespace

Nodes can have attributes which add meta-data to a node. DataModel attributes must not be confused with XML-Attributes. The attributes here are added to a node via special XML-tags.

Attributes have a name and can have multiple string values.

Attributes come in two forms:

1. passive attributes
2. generic attributes
3. automatic attributes

5.8.4.1. Passive attributes

Passive attributes are attributes for which the values are defined in the XDM-file:

```
'<a:da name="' attribute-name '" value="' attribute-value '"/>'

'<a:da name="' attribute-name '">'
  '<a:v>' value '</a:v>'
  ...
'</a:da>'
```

In case of multiple values, or if the value contains line-feeds, the values are set via the `a:v` tag.

5.8.4.2. Generic attributes

So-called generic attributes are able to store any (also quite complex) data structures. There is an own namespace for every attribute-type inside the a:a tag and therefore each generic attribute looks different. Thus, all currently defined generic attributes are explained in the following.

5.8.4.2.1. AdminData

The generic attribute "ADMIN-DATA" is especially used to hold the complex data-structures of that attribute defined by Autosar in a generic way. For this, all TAGs and XML-attributes are taken over from the Autosar format and prefixed with the namespace "ad" in the following form:

```
<a:a name="ADMIN-DATA" type="ADMIN-DATA">
  <ad:ADMIN-DATA>
    <ad:LANGUAGE>DE</ad:LANGUAGE>
    <ad:USED-LANGUAGES>
      <ad:L-10 L="FR">francais</ad:L-10>
    </ad:USED-LANGUAGES>
    <!-- see Autosar format -->
  </ad:ADMIN-DATA>
</a:a>
```

Instead of the comment shown above, there may be more tags as defined by Autosar.

5.8.4.2.2. Implementation Config Class

The attribute IMPLEMENTATIONCONFIGCLASS holds the class and the variant as value as shown below:

```
<a:a name="IMPLEMENTATIONCONFIGCLASS" type="IMPLEMENTATIONCONFIGCLASS">
  <icc:v class="PreCompile">VariantPreCompile</icc:v>
  <icc:v class="Link">VariantLinkTime</icc:v>
  <icc:v class="PostBuild">VariantPostBuild</icc:v>
</a:a>
```

5.8.4.3. Automatic attributes

Automatic attributes calculate their own values by querying the configuration stored in the DataModel. Automatic attributes have a type that determines the algorithm via which the values are calculated and they are equipped with test expressions that describe how to calculate the values.

Automatic attributes are created with the following XML-tags:

```
'<a:da name="' attribute-name '" '
      'type="' attribute-type " '
      'expr="' test-expression '" '
      [ 'true="' true-value '" ' ]
      [ 'false="' false-value '" ' ]
      '/>'

'<a:da name="' attribute-name '" '
      'type="' attribute-type " '
      '>'
  '<a:tst expr="' test-expression '" '
      [ 'true="' true-value '" ' ]
      [ 'false="' false-value '" ' ]
      '/>'
  ...
'</a:da>'
```

The second form allows to define multiple test-expressions.

The Attribute type is mapped via the current DataModelFactory to an implementation. The type-attribute determines the type of the Automatic Attribute. The syntax of the XML-attribute test expression is determined by the type of the automatic attribute.

Each Automatic Attribute has one to n tests. A test is composed of a test expression and an optional fixed true value and an optional fixed false value. Each test evaluates to either no value or a list of values depending on the test expression. The test expression is evaluated by the attribute each time a value is requested from the attribute. If either `true` or `false` or both are set, the result of the expression is interpreted as a boolean value. If the expression evaluates true, the value given with the `true`-attribute is returned. Otherwise the value of the `false`-attribute is returned. If the corresponding attribute is missing, no value is returned. If both, the `true` and the `false`-attributes are not set, the value of the expression is returned. The value of the whole Automatic Attribute is the list of all values returned by all individual tests.

By default the following types are supported:

- Bound
- Match
- Range
- XPath

5.8.4.3.1. Bound attribute

The `Bound` attribute is the simplest form of an Automatic Attribute. The expression always evaluates the values of another node in the tree (the value of a variable, the member names of a map, the indices in a list, the variable names of a container). The bound node is referred to by an absolute or relative XPath expression.

Example:

```
<v:var name="Int1" type="INTEGER"/>
<v:var name="Int2" type="INTEGER"/>
  <a:da name="DEFAULT" type="Bound" expr="../Int1"/>
</v:var>
```

The DEFAULT-value of node Int2 is now the value of Int1.

5.8.4.3.2. Match attribute

The second form, the `Match` Attribute is a slightly more generic form of the `Bound`-attribute. It allows to compare the value of a node referred to by an XPath expression to a fixed value. The expression obeys the rules of a full XPath expression:

```
xPath-to-the-variable compare-operator 'string-value'
xPath-to-the-variable compare-operator number-value
```

The variable to be bound to is selected with an XPath. The value of the variable is matched to the value following the operator. If the value is enclosed in single quotes, it is interpreted as a string. Otherwise it is interpreted as a number.

If the value is a string the following operators are supported:

=
: Returns true if the value of the XPath matches the value in the test.

!=
: Returns true if the value of the XPath does not match the value in the test.

If the value is a number the following operators are supported:

=
: Returns true if the value of the XPath matches the value in the test.

!=
: Returns true if the value of the XPath does not match the value in the test.

<
: Returns true if the value of the XPath is smaller than the value in the test.

<=
: Returns true if the value of the XPath is smaller or equal than the value in the test.

>
: Returns true if the value of the XPath is greater then the value in the test.

>=
: Returns true if the value of the XPath is greater or equal then the value in the test.

Example:

```
<a:da name="TITLE" type="Match">
  <a:tst expr="//Var1 = 3"
```

```
            true="Do not enter something"
            false="Do enter something"/>
    </a:da>
```

The TITLE of the node which owns this attribute depends on the value of another node (//Var1).

5.8.4.3.3. Range attribute

The `Range`-attribute is used to define range checks. The Range attributes works similar to the Match attribute. It defines value range that the value of the node with the `Range`-attribute may have. If one of the range checks does not match the `Range`-attribute produces a corresponding error message as its value. The following expressions are allowed:

=Constant
: Returns true if the value of the node matches the value of the constant.

!=Constant
: Returns true if the value of the XPath does not match the Constant.

<Constant
: Returns true if the value of the XPath is smaller than the value in the test.

<=Constant
: Returns true if the value of the node is smaller or equal as the Constant.

>Constant
: Returns true if the value of the node is greater then the value of the Constant.

>=Constant
: Returns true if the value of the node is greater or equal then the Constant.

Constants may be either numbers which are then interpreted as 64bit float values or if enclosed in `i()` as an integer.

e.g.

```
<a:da name="INVALID" type="Range">
  <a:tst expr="&gt;=i(15)"/>
  <a:tst expr="&lt;=i(30)"/>
</a:da>
```

The example shows the use of the `Range`-attribute. It states that the value of the node must be between 15 and 30.

Note

Do not confuse automatic attributes of type "Range" with the attribute "RANGE" which restricts the valid values of its node.

5.8.4.3.4. XPath attribute

The fourth predefined Automatic Attribute is the `XPath` type. With this type the full power of the `XPath` language can be used (not only Path expressions as in the previous cases). The result of the XPath evaluation is used as the Attribute value. Caution: This is the slowest form of Automatic Attribute.

Example:

```
<a:da name="EDITABLE" type="XPath" expr="../Integer2 &lt; 20"/>
```

The node now is only editable, if the value of node "Integer2" is lower than 20.

5.8.4.3.5. XDM attribute examples

All examples described here are contained in the installation in the demos-directory. Simply copy the Custom-GeneratorModule-directory into the plugin-directory and restart . Then, if you create a project with target "V850/V850EGP1" and add the Module Configuration "Custom Generator Module V850", you can see the attributes "in action".

Check the value of another node

```
<v:var name="OnlyFor16MHz" type="INTEGER">
  <a:da name="ENABLE" type="XPath" expr="../Frequency = '16MHz'"
        true="true" false="false"/>
</v:var>
```

Only enables the node if the node 'Frequency' contains the value '16MHz'.

Negation

```
<v:var name="Negation" type="INTEGER">
  <a:da name="INVALID" type="XPath" expr="not(. = ../OnlyFor16MHz)"
        false="May not be the same value as OnlyFor16MHz"/>
</v:var>
```

Marks the node as invalid, if its value is not the same as 'OnlyFor16MHz'. In this case, the "false"-value is given to the user as an error.

Logical Operators: and, or

```
<v:var name="BiggerThanMHz" type="INTEGER">
  <a:da name="INVALID" type="XPath">
    <a:tst expr="(../Frequency='8MHz' and . &gt; 8)
                 or (../Frequency='16MHz' and . &gt; 16)
                 or (../Frequency = '32MHz' and . &gt; 32)"
           false="Value must be &gt; the configured MHz"/>
  </a:da>
</v:var>
```

Restricts the value of the node depending on the string-value of another node. If the value violates the restriction, the "false"-value is given to the user as error.

Range-check using the Match-attribute

```
<v:var name="SmallIntegerOnly" type="INTEGER">
  <a:da name="INVALID" type="Match">
    <a:tst expr=". &gt;= 1" false="Value must be &gt;=1 and &lt;=9"/>
    <a:tst expr=". &lt;= 9" false="Value must be &gt;=1 and &lt;=9"/>
  </a:da>
</v:var>
```

Marks the node as invalid, if its value is not >=1 and <=9. In this case, the "false"-value is returned as an error.

Range-check using the XPath-attribute

```
<v:var name="NegativeOrBigIntegerOnly" type="INTEGER">
  <a:da name="INVALID" type="XPath"
        expr=". &lt; 0 or . &gt; 9"
        false="Value must be &lt;0 or &gt;9"/>
</v:var>
```

Marks the node as invalid, if its value is not <0 or >9. In this case, the "false"-value is given to the user as error.

Only allow values configured in another list

```
<v:var name="OnlyValuesFromList" type="INTEGER">
  <a:da name="INVALID" type="XPath"
        expr="count(text:grep(../PreConfiguredValues/*, concat('^',.,'$')))>0"
        false="This node may only have values configured in list
               'PreConfiguredValues'"/>
</v:var>
```

Only the value configured within the list 'PreConfiguredValues' are valid values for this node. If another value is given, the "false"-value is given to the user as error.

DEFAULT-value using the Bound-Attribute

```
<v:var name="DefaultToOnlyValuesFromList" type="INTEGER">
  <a:da name="DEFAULT" type="Bound" expr="../OnlyValuesFromList"/>
</v:var>
```

Sets the DEFAULT-value of this node to the value configured for the node 'OnlyValuesFromList'. A "Calc"-button is shown to the user so that he can re-calculate the value on demand since it is only calculated once automatically.

Only allowing certain values using the Range-Attribute

```
<v:var name="Only2or4or8or20" type="INTEGER">
  <a:da name="INVALID" type="Range">
    <a:tst expr="=2" false="Only the values 2, 4, 8 and 20 are allowed"/>
    <a:tst expr="=4" false="Only the values 2, 4, 8 and 20 are allowed"/>
    <a:tst expr="=8" false="Only the values 2, 4, 8 and 20 are allowed"/>
    <a:tst expr="=20" false="Only the values 2, 4, 8 and 20 are allowed"/>
  </a:da>
</v:var>
```

Only allows certain values for the node. If another value is configured, the "false"-value is given to the user as error.

Modulus operator

```
<v:var name="DivisibleBy3" type="INTEGER">
  <a:da name="INVALID" type="XPath" expr=". mod 3 = 0"
        false="Value must be divisible by 3"/>
</v:var>
```

Marks the node as invalid, if the configured value is not divisible by 3. In this case, the "false"-value is given to the user as error.

Calculating with node-values

```
<v:var name="Mul2" type="INTEGER">
  <a:da name="INVALID"
        type="XPath"
        expr=". * ../Mul1 &gt;= 5 and . * ../Mul1 &lt;= 25"
        false="Mul1 * Mul2 must be between 5 and 25"/>
</v:var>
```

The value of the node multiplied with the value of node 'Mul1' and must be between 5 and 25. If not, the "false"-value is given to the user as error.

```
<v:var name="Mul3" type="INTEGER">
  <a:da name="INVALID"
        type="XPath"
        expr=". * ../Mul1 = 5
           or . * ../Mul1 = 10
           or . * ../Mul1 = 20
           or . * ../Mul1 = 40"
        false="Mul1 * Mul3 must be one of 5 10 20 or 40"/>
</v:var>
```

The value of the node multiplied with the value of node 'Mul1' must be 5, 10, 20 or 40. If not, the "false"-value is given to the user as error.

Regular expressions

```
<v:var name="Regex" type="STRING">
  <a:da name="INVALID" type="XPath"
        expr="text:grep(., '^[A-Z][a-zA-Z0-9]*$')"
        false="Must start with a capital letter"/>
</v:var>
```

Marks the value of the node as invalid, if it does not match a regular expression. In this case, the "false"-value is given to the user as error.

References

```
<v:ref name="Reference" type="REFERENCE">
  <a:da name="REF" value="ASPathDataOfSchema:/AUTOSAR/Adc/Container/Int"/>
  <a:da name="RANGE" type="XPath">
    <a:tst expr="text:difference(
      node:paths(node:refs('ASPathDataOfSchema:/AUTOSAR/Adc/Container/Int')),
      node:paths(node:refs(node:refs(
        'ASPathDataOfSchema:/AUTOSAR/Adc/Container/Reference4'
      )))
    )"/>
  </a:da>
```

```
  <a:da name="INVALID"
        type="XPath"
        expr="node:refvalid(.)"
        false="The configured node does not exist or may not be referenced."/>
</v:ref>
```

Both, the REF- and the RANGE-attribute can be used to define which nodes are valid to be configured by the user. If none of these attributes is given, any node can be configured. If both attributes are given, the intersection of the returned values is calculated.

The REF-attribute may only hold paths (no xpath-expression). In the example above, all configured values (DataNodes) for the SchemaNode with the Autosar ShortName-path "/AUTOSAR/Adc/Container/Int" are allowed to be configured.

The RANGE-attribute is an automatic attribute and thus may contain complex xpath-expressions. The example above does not allow to configure a node which is referenced by a ReferenceNode of SchemaNode "/AUTOSAR/Adc/Container/Reference4".

The INVALID-attribute in the example above creates an error, if the node does not reference a valid node (defined by REF and/or RANGE). By default, only a warning is created in that case.

Additional Explanation: To get the nodes of a prefixed path (e.g. ASPathDataOfSchema), the path must be surrounded by apostrophes and give in to the function "node:ref". To get the target-nodes of a ReferenceNode, also the function "node:ref" must be used. Thus, the example above uses this function twice. Because we do not need nodes, but paths, the function "node:paths" must be used. Finally, we have two lists of paths from which we need the "remaining quantity". This is done by "text:difference". Alternatively you could use "node:difference" to the list of nodes and surround it by "node:paths".

Preconfigured values for list-entries

```
<v:lst name="PreConfiguredValues">
  <v:var name="Int" type="INTEGER">
    <a:da name="DEFAULT" type="XPath" expr=
"text:split('1 2 4 8 10', ' ')[position()-1 = node:current()/@index]"
    />
  </v:var>
</v:lst>
```

The DEFAULT-value of each value added to the list depends on its index within the list. By this several preconfigured values can be defined.

5.8.4.3.6. Restrictions

There are some things you cannot do with automatic attributes in the DataModel:

- There is no way to calculate a value of a node all the time, so it is always up to date. Instead, you can use the DEFAULT-attribute which gives the user the possibility to recalculate the value of the node. For this all nodes with an automatic DEFAULT attribute have a "Calculate"-button.
- You cannot use the attribute LABEL as automatic attribute. The LABEL of an entry-field always has a static value. The only automatic value may be the TITLE of the page.

5.8.4.4. Data attributes

In addition to attributes of a node schema-nodes can have so called *data-attributes*. Data attributes are attributes that are inherited by data-nodes that are bound to the schema-node.

Automatic data attributes are therefore never evaluated in the context of the schema-node but only in the context of the data-node (essential for relative path expression).

Data attributes are defined like normal attributes but with the XML-tag `a:da`.

Example:

```
<v:ctr name="myCtrDef">
  <a:da name="ENABLE" type="XPath" expr="do_enable = 'TRUE'"/>
  </a:da>
</v:ctr>

...

<d:ctr name="myCtrData">
  <a:a name="DEF" value="XPath:/.../myCtrDef"/>

  <d:var name="do_enable" value="FALSE"/>
</d:ctr>
```

In the example above the `ENABLE` data attribute is evaluated in the context of `myCtrData` and therefore evaluates 'false' as the variable `do_enable` is set to false.

5.8.4.5. Common attributes

All nodes can be equipped with the following attributes:

Attribute	Type	Description
IMPORTER_INFO	string array	Used by importers to add information from where a node was imported.

Table 5.5. Attributes

5.8.5. Generic nodes

Generic nodes are neither data nor schema-nodes.

Feature	Description
Namespace	http://www.tresos.de/_projects/DataModel2/06/root.xsd
Default Prefix	None

Table 5.6. Namespace

The DataModel defines a single generic node type:

5.8.5.1. link

Link nodes are proxies for other nodes that are referred to by the link node. Link nodes can be used to define schema-nodes that, for example, contain themselves:

```
<v:ctr name="myContainer">
  <link name="subContainer" value="XPath:.."/>
</v:ctr>
```

5.8.5.1.1. Attributes

None

5.8.5.1.2. Valid children

None

5.8.6. Schema-nodes

Schema-nodes define available configuration parameters. Schema-nodes form trees that are referred to (bound) by data-node trees that conform to the schema. Schema-nodes can have data attributes in addition to normal attributes. Schema-nodes always have a unique name concerning their parent. Schema-nodes never have a value.

Feature	Description
Namespace	http://www.tresos.de/_projects/DataModel2/06/schema.xsd
Default Prefix	v

Table 5.7. Namespace

Data attributes that can be used for all schema-nodes:

Attribute	Type	Description
ENABLE	boolean	If set to false the data-node is treated as not being part of the tree.
INVALID	string array	This must be an automatic attribute that evolves a list of error messages. If so the data-node is considered as being erroneous.
WARNING	string array	This must be an automatic attribute that evolves a list of warning messages. If so the data-node is considered as being possibly incorrect.
DESC	string	Description of the node (e.g. for tool tips).
ORIGIN	string	Description of where the parameter originates (vendor, module name, version, target).

Table 5.8. Common data attributes

5.8.6.1. ctr

The `v:ctr` define a container schema-node. A container groups (contains) a fixed list of other nodes all identified by a unique name.

5.8.6.1.1. Attributes and data attributes

Attribute	Type	Description
NAME_PATTERN	String	The optional attribute `NAME_PATTERN` affects the naming of a newly created instance of the container. If this attribute is not explicitly set, a default name pattern is created for any container consisting of the container's name and a unique index. The value of the attribute must consist of a valid node name. A valid AUTOSAR node name begins with a letter and may be followed by letters, numbers and underscore. The maximum length of the name must not exceed 32 characters (Regular expression: `^[a-zA-Z][a-zA-Z_0-9]{0,31}$`). The value of attribute `NAME_PATTERN` must additionally contain a question mark unless the container can only be created once. The question mark will be replaced with an incremetal index by the system to assure the uniqueness of the name. Please note, that the container cannot be created if the resulting name is not unique. Example: `MyContainerName_?`, `My?Container`

Table 5.9. Schema attributes

5.8.6.1.2. Valid children

- v:ctr
- v:chc
- v:lst
- v:var
- v:ref

5.8.6.2. chc

A `v:chc` defines a choice schema-node. A choice schema node contains container schema-nodes. In the configuration only one container data-node can be added to a choice that must conform to one of the containers of the choice schema-node. A choice is therefore an element with which the user can select different alternatives for the element representation.

5.8.6.2.1. Attributes and data attributes

None

5.8.6.2.2. Valid children

- v:ctr

5.8.6.3. lst

A `v:lst` creates a list schema-node. A list in the schema has only one child node which can be replicated in the data multiple times. List schema-nodes can be used to define tables.

Lists come in two variations: lists and maps which are selected via the `TYPE`-attribute of the list schema-node:

""
: The empty type creates a list node. A list just contains an ordered list of entries.

"MAP"
: A list of the type MAP creates a map. A map is an ordered list where the names of all entries are unique within the list.

Available Data attributes in addition to the ones that can be used for all schema-nodes:

5.8.6.3.1. Attributes and data attributes

Attribute	Type	Description
MIN	long	Defines the minimum number of nodes in the (data-node) list. It is an error if the list contains fewer entries. When the (data-node) list is created it has a `MIN` number of entries.
MAX	long	Defines the maximum number of nodes in the (data-node) list. It is an error if the list contains more entries.

Table 5.10. Data attributes

5.8.6.3.2. Valid children

- v:ctr
- v:chc
- v:var

- v:ref

5.8.6.4. var

A `v:var` creates a variable schema-node. A variable schema node defines a configuration parameter that can store a single value.

The type of the value is determined by the `type` XML-attribute of the `v:var`-tag. The following types are supported:

INTEGER:
: Can store a 64 bit signed integer. Integer values are encoded in decimal format.

BOOLEAN:
: Stores a boolean value. The encoding of true and false is determined by the `RANGE` attribute. The first value of the `RANGE`-attribute defines the true value, the second defines the false value. True defaults to the string "TRUE", false defaults to "FALSE".

FLOAT:
: Stores a 64 bit signed float value. Float values are encoded in decimal format (`[ "-" ] <digits> "." <digits>`).

STRING:
: Stores an arbitrary ASCII string.

5.8.6.4.1. Attributes and data attributes

Attribute	Type	Description
DEFAULT	string	Defines the default value of the data-node. This may also be an automatic attribute.

RANGE	string array	Restricts valid value ranges. The interpretation of the range depends on the type of the schema-node.
		For the types `INT` and `FLOAT`, the `RANGE` can restrict the values with the following expressions:
		`[ '-' ] number '-' [ '-' ] number`
		`'<' [ "-" ] number`
		`'<=' [ '-' ] number`
		`'>' [ '-' ] number`
		`'>=' [ '-' ] number`
		Type `STRING` variables can be restricted by either a list of values or by a regular expression. A regular expression must be prefixed with ~ (tilde).
		Type `BOOLEAN` variables define true and false values with their range attribute. The first value of the range attribute is the true value whereas the second is the false value.

Table 5.11. Data attributes

5.8.6.4.2. Valid children

none

5.8.6.5. ref

A `v:ref` creates a reference schema-node. A reference schema node defines a configuration parameter that refers to another configuration parameter.

5.8.6.5.1. Attributes and data attributes

Attribute	Type	Description
REF	string array	A list of paths that point to schema-nodes. Data-nodes that refer to these schema-nodes can be selected.
RANGE	string array	A list of paths that point to schema-nodes. Data-nodes that refer to these schema-nodes can be selected. This is usually an automatic attribute which calculates the nodes. For examples see Section 5.8.4.3.5: XDM attribute examples.

Table 5.12. Data attributes

If none of these attributes is given, all nodes can be selected. If both attributes are given, the intersection of the returned values is used to calculate the nodes which can be selected.

5.8.6.5.2. Valid children

none

5.8.7. Data-nodes

Data-nodes store configuration data. Data-nodes form trees that refer to (or bind) schema-nodes which describe their data ranges etc.

Feature	Description
Namespace	http://www.tresos.de/_projects/DataModel2/06/data.xsd
Default Prefix	d

Table 5.13. Namespace

Attributes that can be used for all data-nodes:

Attribute	Type	Description
DEF	string	Node reference to the schema-node that describes the data-node.

Table 5.14. Common Attributes

5.8.7.1. ctr

The `d:ctr` defines a container data-node. A container groups (contains) a fixed number of other nodes all identified by a unique name.

5.8.7.1.1. Attributes

none

5.8.7.1.2. Valid Children

- d:ctr
- d:chc
- d:lst
- d:var
- d:ref

5.8.7.2. chc

A `d:chc` defines a choice data-node. A choice data node contains one container node. The container data-node must conform to one of the container schema-nodes of the choice schema node to which the choice data-node is bound.

5.8.7.2.1. Attributes

none

5.8.7.2.2. Valid Children

- d:ctr

5.8.7.3. lst

A `d:lst` creates a list data-node. A list in the data has children of a single type (container, choice, variable, reference). The children of a list data-node must conform to the child of the bound schema node.

Lists come in two variations: lists and maps which are selected via the `TYPE`-attribute of the list schema-node:

""
: The empty type creates a list node. A list contains an ordered list of entries.

"MAP"
: A list of the type MAP creates a map. A map is an ordered list where the names of all entries are unique within the list.

5.8.7.3.1. Attributes

none

5.8.7.3.2. Valid Children

- d:ctr
- d:chc
- d:var
- d:ref

5.8.7.4. var

A `d:var` creates a variable data-node. A variable data node stores a single configuration value.

5.8.7.4.1. Attributes

none

5.8.7.4.2. Valid Children

none

5.8.7.5. ref

A `d:ref` creates a reference data-node. A reference data node is a configuration parameter that refers to another configuration parameter. The reference to that node is stored in the value of the data-node as a node path expression.

5.8.7.5.1. Attributes

none

5.8.7.5.2. Valid Children

none

5.8.8. Factories

The DataModel uses factories to create nodes. Each factory has knowledge of different node types and may tailor created nodes (add specific attributes etc.). Additionally, each factory may define new namespaces for nodes and attributes defined by this factory.

The following factories are supported:

default:
: A factory that creates default node instances for all node types.

autosar:
: This factory must be used to create nodes for the Autosar ECU Configuration, ECU Parameter Definition and the other Autosar formats.

The factory of a node can be set with the XML-Attribute `factory` which can be applied to all node tags. When a node has a factory-attribute, the named factory is used to create the node and all subsequent nodes that do not have such a tag, i.e. if the same factory shall be used for all nodes, it only has to be specified for the root-node.

If a factory defines attributes or nodes with a special namespace, the namespace(s) must be defined at the node which specifies the factory, e.g.:

```
<d:ctr type="AUTOSAR" factory="autosar"
  xmlns:ad="http://www.tresos.de/_projects/DataModel2/08/admindata.xsd"
  xmlns:icc="http://www.tresos.de/_projects/DataModel2/08/implconfigclass.xsd">
```

5.8.9. Path addressing

Reference data-nodes, the `DEF` attribute of data-nodes and the `REF` data attribute of reference schema-nodes refer to other nodes in their string values. This is be done by a path expression of the form:

```
path-type ':' path
```

The `DEF`-attribute of a data-node therefore might look like the following with XPath addressing:

```
<d:ctr name="myCtr">
  <a:a name="DEF" value="XPath:/AUTOSAR/
  TOP-LEVEL-PACKAGES/myPkg/ELEMENTS/myModuleDef/myCtr"/>
</d:ctr>
```

or with Autosar addressing:

```
<d:ctr name="myCtr">
  <a:a name="DEF" value="ASPath:/myPkg/myModuleDef/myCtr"/>
</d:ctr>
```

The following addressing schemas are supported:

XPath:
: References a node via an XPath path (not a full XPath expression).

XPathDataOfSchema:
: References all data-nodes that are bound to the schema node to which the XPath expression points.

SchemaViaParentByIdx:
: Takes the schema-node of the parent node and returns the schema-child of the node that has the value of the expression as it's index. If expression is empty a value of 0 is used.

SchemaViaParent:
: Takes the schema-node of the parent node and returns the schema-child of the node that has the value of the expression as it's name. If expression is empty the node-name is used. If the parent is no list node, the first schema-parent-child is used.

ASPath:
: Refers to a node by an Autosar path expression. This path expression is only available for nodes that were created by the Autosar factory.

ASPathDataOfSchema:
: Refers to all data-nodes that are bound to the schema node to which the Autosar path expression points. This path expression is only available for nodes that were created by the Autosar factory.

ASPathParentNode:
: References the parent node of a node that is referred to by an Autosar path. This path expression is only available for nodes that were created by the Autosar factory.

ASPathDataOfSchemaChoice:
: Takes the schema-node of the parent node and returns the schema-child of the node that has the value of the expression as it's name. If expression is empty the node-name is used. If the parent is no list node, the first schema-parent-child is used. This path expression is only available for nodes that were created by the Autosar factory.

5.9. Mapping Autosar ECU Configuration to XDM

5.9.1. Node mapping

The ECU Parameter Definition will be represented as a schema-tree within the DataModel and the ECU Configuration Description as a data-tree. All Autosar-nodes will be mapped to the nodes of the DataModel:

5.9.1.1. Schema-Nodes

5.9.1.1.1. AR-PACKAGE

There is no corresponding data node.

simple
`<d:ctr type="AR-PACKAGE"/>`

5.9.1.1.2. MODULE-DEF

Corresponding data node: Section 5.9.1.13.2: MODULE-CONFIGURATION.

simple
`<v:ctr name="myModule" type="MODULE-DEF"/>`

5.9.1.1.3. PARAM-CONF-CONTAINER-DEF

Corresponding data node: Section 5.9.1.13.3: CONTAINER.

simple	with surrounding list

`<v:ctr name="myCtr"`
`  type="IDENTIFIABLE"/>`
	`<v:lst name="myCtr" type="MAP">`
	`  <v:ctr name="myCtr"`
	`  type="IDENTIFIABLE"/>`
	`</v:lst>`

5.9.1.1.4. MULTIPLE-CONFIGURATION-CONTAINER

The tag MULTIPLE-CONFIGURATION-CONTAINER may only be contained within a PARAM-CONF-CONTAINER-DEF and marks this container as multiple configuration set. This information cannot be seen in the data-tree. Within the xdm file, there is always a list within the schema and data-tree! Corresponding data node: Section 5.9.1.13.3: CONTAINER.

with surrounding list

`<v:lst name="DioConfiguration" type="MULTIPLE-CONFIGURATION-CONTAINER">`
`  <a:da name="MIN" value="1"/>`
`  <v:ctr name="DioConfiguration" type="MULTIPLE-CONFIGURATION-CONTAINER">`
`  ...`
`  </v:ctr>`
`</v:lst>`

5.9.1.1.5. BOOLEAN-PARAM-DEF

Corresponding data node: Section 5.9.1.13.5: BOOLEAN-VALUE.

simple	with surrounding list

`<v:var name="bit" type="BOOLEAN"/>`	`<v:lst name="bit">`
	`  <v:var name="bit" type="BOOLEAN"/>`
	`</v:lst>`

5.9.1.1.6. INTEGER-PARAM-DEF

Corresponding data node: Section 5.9.1.1.7: FLOAT-PARAM-DEF.

simple	with surrounding list

`<v:var name="myInt" type="INTEGER"/>`	`<v:lst name="myInt">`
	`  <v:var name="myInt" type="INTEGER"/>`
	`</v:lst>`

5.9.1.1.7. FLOAT-PARAM-DEF

Corresponding data node: Section 5.9.1.13.7: FLOAT-VALUE.

simple	with surrounding list

`<v:var name="myFloat" type="FLOAT"/>`	`<v:lst name="value">`
	`  <v:var name="value" type="FLOAT"/>`
	`</v:lst>`

5.9.1.1.8. STRING-PARAM-DEF

Corresponding data node: Section 5.9.1.13.8: STRING-VALUE.

simple	with surrounding list

`<v:var name="myStrg" type="STRING"/>`	`<v:lst name="myStrg">`
	`  <v:var name="myStrg" type="STRING"/>`
	`</v:lst>`

5.9.1.1.9. LINKER-SYMBOL-DEF

Corresponding data node: Section 5.9.1.13.9: LINKER-SYMBOL-VALUE.

simple	with surrounding list

`<v:var name="LinkerSymbol"`
`  type="LINKER-SYMBOL"/>`
	`<v:lst name="LinkerSymbol">`
	`  <v:var name="LinkerSymbol"`
	`  type="LINKER-SYMBOL"/>`
	`</v:lst>`

5.9.1.1.10. FUNCTION-NAME-DEF

Corresponding data node: Section 5.9.1.13.10: FUNCTION-NAME-VALUE.

simple	with surrounding list

`<v:var name="myFunction"`
`  type="FUNCTION-NAME"/>`
	`<v:lst name="myFunction">`
	`  <v:var name="myFunction"`
	`  type="FUNCTION-NAME"/>`
	`</v:lst>`

5.9.1.1.11. ENUMERATION-PARAM-DEF

Corresponding data node: Section 5.9.1.13.11: ENUMERATION-VALUE.

simple	with surrounding list

`<v:var name="enum"`
`  type="ENUMERATION">`
`  <a:da name="RANGE">`
`  <a:v>Enum1</a:v>`
`  <a:v>Enum2</a:v>`
`  </a:da>`
`</v:var>`
	`<v:lst name="enum">`
	`  <v:var name="enum"`
	`  type="ENUMERATION">`
	`  <a:da name="RANGE">`
	`  <a:v>Enum1</a:v>`
	`  <a:v>Enum2</a:v>`
	`  </a:da>`
	`  </v:var>`
	`</v:lst>`

5.9.1.2. CHOICE-CONTAINER-DEF

Up to Autosar 2.1, there is no corresponding TAG for a data node in Autosar. Nevertheless there is a data node in tresos: Section 5.9.1.13.4: Choices. Since Autosar 3.0, there is a corresponding Data-Node-tag which has the same tag as containers (CONTAINER).

simple	with surrounding list

Autosar 2.x:
`<v:chc name="myChc">`
`  <v:ctr name="myCtr1"/>`
`  <v:ctr name="myCtr2"/>`
`</v:chc>`
	`<v:lst name="myChc" type="MAP">`
	`  <v:chc name="myChc">`
	`  <v:ctr name="myCtr1"/>`
	`  <v:ctr name="myCtr2"/>`
	`  </v:chc>`
	`</v:lst>`

Autosar 3.0:
`<v:chc name="myChc"`
`  type="IDENTIFIABLE">`
`  <v:ctr name="myCtr1"/>`
`  <v:ctr name="myCtr2"/>`
`</v:chc>`
	`<v:lst name="myChc" type="MAP">`
	`  <v:chc name="myChc"`
	`  type="IDENTIFIABLE">`
	`  <v:ctr name="myCtr1"/>`
	`  <v:ctr name="myCtr2"/>`
	`  </v:chc>`
	`</v:lst>`

5.9.1.3. CHOICE-REFERENCE-DEF

Corresponding data node: Section 5.9.1.13.12: REFERENCE-VALUE.

simple	with surrounding list

`<v:ref name="myChrRef"`
`  type="CHOICE-REFERENCE">`
`  <a:da name="REF"`
`  value="ASPathDataOfSchema:/..."/>`
`  <a:da name="REF-TYPE"`
`  value="ASPath"/>`
	`<v:lst name="myChrRef">`
	`  <v:ref name="myChrRef"`
	`  type="CHOICE-REFERENCE">`
	`  <a:da name="REF"`
	`  value="ASPathDataOfSchema:/..."/>`
	`  <a:da name="REF-TYPE"`

simple	with surrounding list

`</v:ref>`	`value="ASPath"/>`
	`  </v:ref>`
	`</v:lst>`

5.9.1.4. REFERENCE-DEF

Corresponding data node: Section 5.9.1.13.12: REFERENCE-VALUE.

simple	with surrounding list

`<v:ref name="myRef" type="REFERENCE">`
`  <a:da name="REF"`
`  value="ASPathDataOfSchema:/..."/>`
`  <a:da name="REF-TYPE"`
`  value="ASPath"/>`
`</v:ref>`
	`<v:lst name="myRef">`
	`  <v:ref name="myRef" type="REFERENCE">`
	`  <a:da name="REF"`
	`  value="ASPathDataOfSchema:/..."/>`
	`  <a:da name="REF-TYPE"`
	`  value="ASPath"/>`
	`  </v:ref>`
	`</v:lst>`

5.9.1.5. SYMBOLIC-NAME-REFERENCE-DEF

Corresponding data node: Section 5.9.1.13.12: REFERENCE-VALUE.

simple	with surrounding list

`<v:ref name="mySrd"`
`  type="SYMBOLIC-NAME-REFERENCE">`
`  <a:da name="REF"`
`  value="ASPathDataOfSchema:/..."/>`
`  <a:da name="REF-TYPE"`
`  value="ASPath"/>`
`</v:ref>`
	`<v:lst name="mySrc">`
	`  <v:ref name="mySrd"`
	`  type="SYMBOLIC-NAME-REFERENCE">`
	`  <a:da name="REF"`
	`  value="ASPathDataOfSchema:/..."/>`
	`  <a:da name="REF-TYPE"`
	`  value="ASPath"/>`
	`  </v:ref>`
	`</v:lst>`

5.9.1.6. INSTANCE-REFERENCE-DEF

Corresponding data node: Section 5.9.1.13.13: INSTANCE-REFERENCE-VALUE.

simple	with surrounding list

`<v:ctr name="myInst" type="INSTANCE">`
`  <v:ref name="TARGET"`
`  type="REFERENCE">`
`  <a:da name="REF"`
`  value="ASTyped:RunnableEntity"/>`
`  </v:ref>`
`  <v:lst name="CONTEXT">`
`  <v:ref name="CONTEXT"`
`  type="REFERENCE">`
`  <a:da name="RANGE"`
`  type="IRefCtxt">`
`  <a:tst expr="ComponentPrototype"/>`
`  <a:tst expr="PortPrototype*"/>`
`  </a:da>`
`  </v:ref>`
`  </v:lst>`
`</v:ctr>`
	`<v:lst name="myInst">`
	`  <v:ctr name="myInst" type="INSTANCE">`
	`  <v:ref name="TARGET"`
	`  type="REFERENCE">`
	`  <a:da name="REF"`
	`  value="ASTyped:RunnableEntity"/>`
	`  </v:ref>`
	`  <v:lst name="CONTEXT">`
	`  <v:ref name="CONTEXT"`
	`  type="REFERENCE">`
	`  <a:da name="RANGE"`
	`  type="IRefCtxt">`
	`  <a:tst expr="ComponentPrototype"/>`
	`  <a:tst expr="PortPrototype*"/>`
	`  </a:da>`
	`  </v:ref>`
	`  </v:lst>`
	`  </v:ctr>`
	`</v:lst>`

5.9.1.7. FOREIGN-REFERENCE-DEF

Corresponding data node: Section 5.9.1.13.12: REFERENCE-VALUE.

simple	with surrounding list

`<v:ref name="myRef"`
`  type="FOREIGN-REFERENCE">`
`  <a:da name="REF"`
`value="ASPathDataOfSchemaChoice:/..."/>`
`  <a:da name="REF-TYPE"`
`  value="ASPath"/>`
`</v:ref>`
	`<v:lst name="myRef">`
	`  <v:ref name="myRef"`
	`  type="FOREIGN-REFERENCE">`
	`  <a:da name="REF"`
	`value="ASPathDataOfSchemaChoice:/..."/>`
	`  <a:da name="REF-TYPE"`
	`  value="ASPath"/>`
	`  </v:ref>`
	`</v:lst>`

5.9.1.8. DERIVED-BOOLEAN-PARAM-DEF

Corresponding data node: Section 5.9.1.13.5: BOOLEAN-VALUE.

simple	with surrounding list

`<v:var name="myBool" type="BOOLEAN">`
`  <a:a name="DERIVED" value="true"/>`
`</v:var>`
	`<v:lst name="myBool">`
	`  <v:var name="myBool" type="BOOLEAN">`
	`  <a:a name="DERIVED" value="true"/>`
	`  </v:var>`
	`</v:lst>`

5.9.1.9. DERIVED-INTEGER-PARAM-DEF

Corresponding data node: Section 5.9.1.13.6: INTEGER-VALUE.

simple	with surrounding list

`<v:var name="myInt" type="INTEGER">`
`  <a:a name="DERIVED" value="true"/>`
`</v:var>`
	`<v:lst name="myInt">`
	`  <v:var name="myInt" type="INTEGER">`
	`  <a:a name="DERIVED" value="true"/>`
	`  </v:var>`
	`</v:lst>`

5.9.1.10. DERIVED-FLOAT-PARAM-DEF

Corresponding data node: Section 5.9.1.13.7: FLOAT-VALUE.

simple	with surrounding list

`<v:var name="myFloat" type="FLOAT">`
`  <a:a name="DERIVED" value="true"/>`
`</v:var>`
	`<v:lst name="myFloat">`
	`  <v:var name="myFloat" type="FLOAT">`
	`  <a:a name="DERIVED" value="true"/>`
	`  </v:var>`
	`</v:lst>`

5.9.1.11. DERIVED-STRING-PARAM-DEF

Corresponding data node: Section 5.9.1.13.8: STRING-VALUE.

simple	with surrounding list

`<v:var name="myStrg" type="STRING">`
`  <a:a name="DERIVED" value="true"/>`
`</v:var>`
	`<v:lst name="myStrg">`
	`  <v:var name="myStrg" type="STRING">`
	`  <a:a name="DERIVED" value="true"/>`
	`  </v:var>`
	`</v:lst>`

5.9.1.12. DERIVED-ENUMERATION-PARAM-DEF

Corresponding data node: Section 5.9.1.13.11: ENUMERATION-VALUE.

simple	with surrounding list

`<v:var name="enum" type="ENUMERATION">`
`  <a:a name="DERIVED" value="true"/>`
`</v:var>`
	`<v:lst name="enum">`
	`  <v:var name="enum"`
	`  type="ENUMERATION">`
	`  <a:a name="DERIVED" value="true"/>`
	`  </v:var>`
	`</v:lst>`

5.9.1.13. Data-Nodes

5.9.1.13.1. ECU-CONFIGURATION

There is no corresponding schema node.

simple

`<d:ctr name="myCfg" type="ECU-CONFIGURATION">`
`  ...`
`</d:ctr>`

5.9.1.13.2. MODULE-CONFIGURATION

Corresponding schema node: Section 5.9.1.1.2: MODULE-DEF.

simple

`<d:ctr name="myMod" type="MODULE-CONFIGURATION">`
`  ...`
`</d:ctr>`

5.9.1.13.3. CONTAINER

Corresponding schema node: Section 5.9.1.1.3: PARAM-CONF-CONTAINER-DEF, Section 5.9.1.1.4: MULTIPLE-CONFIGURATION-CONTAINER.

simple	with surrounding list

`<d:ctr name="myCtr"`
`  type="IDENTIFIABLE">`
`  ...`
`</d:ctr>`
	`<d:lst name="myCtr" type="MAP">`
	`  <d:ctr name="myCtr1"`
	`  type="IDENTIFIABLE1">`
	`  ...`
	`  </d:ctr>`
	`  <d:ctr name="myCtr2"`
	`  type="IDENTIFIABLE2">`
	`  ...`
	`  </d:ctr>`
	`</d:lst>`

5.9.1.13.4. Choices

Autosar 2.0 and 2.1 defines no XML-tag for Choices within the data. They are simply omitted. Since Autosar 3.0, a choice is represented in data as CONTAINER. Nonetheless, there always is a choice in the data-tree in xdm file format. Corresponding schema node: Section 5.9.1.2: CHOICE-CONTAINER-DEF.

Simple	With surrounding list
`Autosar 2.x:`	`<d:lst name="MyChoice" type="MAP">`

Simple	With surrounding list

`<d:chc name="MyChoice"`
`type="CHOICE"`
`value="Ctr1">`
`<d:ctr name="Ctr1"`
`type="IDENTIFIABLE"/>`
`<d:ctr name="Ctr2"`
`type="IDENTIFIABLE"/>`
`<d:ctr name="Ctr3"`
`type="IDENTIFIABLE"/>`
`</d:chc>`
	`<d:chc name="MyChoice"`
	`type="CHOICE"`
	`value="Ctr1">`
	`<d:ctr name="Ctr1"`
	`type="IDENTIFIABLE"/>`
	`<d:ctr name="Ctr2"`
	`type="IDENTIFIABLE"/>`
	`<d:ctr name="Ctr3"`
	`type="IDENTIFIABLE"/>`
	`</d:chc>`
	`</d:lst>`

`Autosar 3.0:`
`<d:chc name="MyChoice"`
`type="IDENTIFIABLE"`
`value="Ctr1">`
`<d:ctr name="Ctr1"`
`type="IDENTIFIABLE"/>`
`<d:ctr name="Ctr2"`
`type="IDENTIFIABLE"/>`
`<d:ctr name="Ctr3"`
`type="IDENTIFIABLE"/>`
`</d:chc>`
	`<d:lst name="MyChoice" type="MAP">`
	`<d:chc name="MyChoice"`
	`type="IDENTIFIABLE"`
	`value="Ctr1">`
	`<d:ctr name="Ctr1"`
	`type="IDENTIFIABLE"/>`
	`<d:ctr name="Ctr2"`
	`type="IDENTIFIABLE"/>`
	`<d:ctr name="Ctr3"`
	`type="IDENTIFIABLE"/>`
	`</d:chc>`
	`</d:lst>`

5.9.1.13.5. BOOLEAN-VALUE

Corresponding schema node: Section 5.9.1.1.5: BOOLEAN-PARAM-DEF, Section 5.9.1.8: DERIVED-BOOLEAN-PARAM-DEF.

simple	with surrounding list
`<d:var name="myBool" value="true"`	`<d:lst name="myBool">`

simple	with surrounding list

`type="BOOLEAN"/>`	`<d:var name="myBool1" value="true"`
	`type="BOOLEAN"/>`
	`<d:var name="myBool2" value="true"`
	`type="BOOLEAN"/>`
	`</d:lst>`

5.9.1.13.6. INTEGER-VALUE

Corresponding schema node: Section 5.9.1.1.6: INTEGER-PARAM-DEF, Section 5.9.1.9: DERIVED-INTEGER-PARAM-DEF.

simple	with surrounding list

`<d:var name="myInt" value="5"`
`type="INTEGER"/>`
	`<d:lst name="myInt">`
	`<d:var name="myInt1" value="5"`
	`type="INTEGER"/>`
	`<d:var name="myInt2" value="5"`
	`type="INTEGER"/>`
	`</d:lst>`

5.9.1.13.7. FLOAT-VALUE

Corresponding schema node: Section 5.9.1.1.7: FLOAT-PARAM-DEF, Section 5.9.1.10: DERIVED-FLOAT-PARAM-DEF.

simple	with surrounding list

`<d:var name="myFloat" value="5.5"`
`type="FLOAT"/>`
	`<d:lst name="myFloat">`
	`<d:var name="myFloat1" value="5.5"`
	`type="FLOAT"/>`
	`<d:var name="myFloat2" value="5.5"`
	`type="FLOAT"/>`
	`</d:lst>`

5.9.1.13.8. STRING-VALUE

Corresponding schema node: Section 5.9.1.1.8: STRING-PARAM-DEF, Section 5.9.1.11: DERIVED-STRING-PARAM-DEF.

simple	with surrounding list

`<d:var name="myStr" value="Hello"`
`type="STRING"/>`
	`<d:lst name="myFloat">`
	`<d:var name="myStr1" value="Hello"`
	`type="STRING"/>`
	`<d:var name="myStr2" value="Hello"`
	`type="STRING"/>`
	`</d:lst>`

5.9.1.13.9. LINKER-SYMBOL-VALUE

Corresponding data node: Section 5.9.1.1.9: LINKER-SYMBOL-DEF.

simple	with surrounding list

`<d:var name="LinkerSymbol"`
`type="LINKER-SYMBOL"`
`value="MyLinkerSymbol"/>`
	`<d:lst name="LinkerSymbol">`
	`<d:var name="LinkerSymbol1"`
	`type="LINKER-SYMBOL"`
	`value="MyLinkerSymbol"/>`
	`<d:var name="LinkerSymbol2"`
	`type="LINKER-SYMBOL"`
	`value="MyLinkerSymbol"/>`
	`</d:lst>`

5.9.1.13.10. FUNCTION-NAME-VALUE

Corresponding schema node: Section 5.9.1.1.10: FUNCTION-NAME-DEF.

simple	with surrounding list
`<d:var name="myFunc" value="func1"`	`<d:lst name="myFunc">` `<d:var name="myFunc1" value="func1"`

simple	with surrounding list

`type="FUNCTION-NAME"/>`	`type="FUNCTION-NAME"/>`
	`<d:var name="myFunc2" value="func1"`
	`type="FUNCTION-NAME"/>`
	`</d:lst>`

5.9.1.13.11. ENUMERATION-VALUE

Corresponding schema node: Section 5.9.1.1.11: ENUMERATION-PARAM-DEF, Section 5.9.1.12: DERIVED-ENUMERATION-PARAM-DEF.

simple	with surrounding list

`<d:var name="myEnum" value="Enum1"`
`type="ENUMERATION"/>`
	`<d:lst name="myEnum">`
	`<d:var name="myEnum1" value="Enum1"`
	`type="ENUMERATION"/>`
	`<d:var name="myEnum2" value="Enum1"`
	`type="ENUMERATION"/>`
	`</d:lst>`

5.9.1.13.12. REFERENCE-VALUE

Corresponding schema node: Section 5.9.1.4: REFERENCE-DEF, Section 5.9.1.5: SYMBOLIC-NAME-REFERENCE-DEF, Section 5.9.1.7: FOREIGN-REFERENCE-DEF, Section 5.9.1.3: CHOICE-REFERENCE-DEF.

simple	with surrounding list

`<d:ref name="myRef"`
`value="ASPath:/..."`
`type="REFERENCE"/>`
	`<d:lst name="myRef">`
	`<d:ref name="myRef1"`
	`value="ASPath:/..."`
	`type="REFERENCE"/>`
	`<d:ref name="myRef2"`
	`value="ASPath:/..."`
	`type="REFERENCE"/>`
	`</d:lst>`

5.9.1.13.13. INSTANCE-REFERENCE-VALUE

Corresponding schema node: Section 5.9.1.6: INSTANCE-REFERENCE-DEF.

simple	with surrounding list

`<d:ctr name="iRef" type="INSTANCE">`
`  <d:lst name="CONTEXT">`
`  <d:ref value="ASPath:/..."`
`  type="REFERENCE"/>`
`  <d:ref value="ASPath:/..."`
`  type="REFERENCE"/>`
`  </d:lst>`
`  <d:ref name="TARGET"`
`  value="ASPath:/..."`
`  type="REFERENCE"/>`
`</d:ctr>`
	`<d:lst name="iRef">`
	`  <d:ctr name="iRef1" type="INSTANCE">`
	`  <d:lst name="CONTEXT">`
	`  <d:ref value="ASPath:/..."`
	`  type="REFERENCE"/>`
	`  <d:ref value="ASPath:/..."`
	`  type="REFERENCE"/>`
	`  </d:lst>`
	`  <d:ref name="TARGET"`
	`  value="ASPath:/..."`
	`  type="REFERENCE"/>`
	`  </d:ctr>`
	`</d:lst>`

5.9.1.13.14. Example:

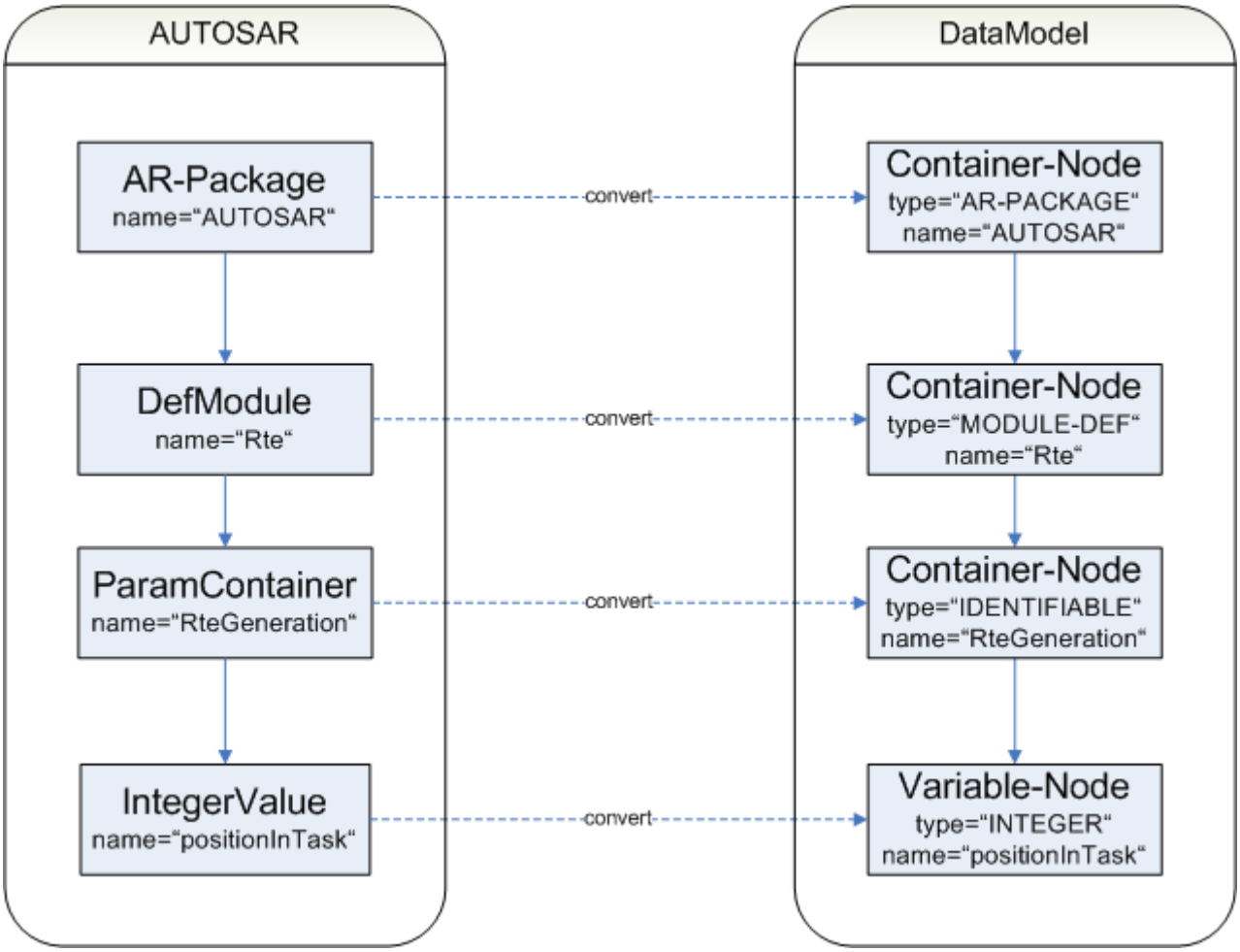


Figure 5.12. Autosar-Modes to DataModel-Nodes

Every ECU Configuration Description-node refers to one ECU Parameter Definition-node. The tree-structure within the DataModel looks like the structure of the ECU-PD/CD except for the top level structure, lists and choices (see the following chapters).

5.9.2. Attribute Mapping

There are some complex attributes in Autosar which have their own mapping.

5.9.2.1. CONFIGURATION-CLASS-AFFECTION

Autosar	XDM

`<CONFIGURATION-CLASS-AFFECTION>`
`  <AFFECTED-REFS>`
`  <AFFECTED-REF`
`  DEST="BOOLEAN-PARAM-DEF">`
`  /abc..`
`  </AFFECTED-REF>`
`  <AFFECTED-REF`
`  DEST="STRING-PARAM-DEF">`
`  /xyz..`
`  </AFFECTED-REF>`
`  </AFFECTED-REFS>`
`  <AFFECTION-KIND>`
`  PC-AFFECTS-PB`
`  </AFFECTION-KIND>`
`</CONFIGURATION-CLASS-AFFECTION>`
	`<a:a name="AFFECTED">`
	`  <a:v>ASPathDataOfSchema:/abc..</a:v>`
	`  <a:v>ASPathDataOfSchema:/xyz..</a:v>`
	`</a:a>`
	`<a:a name="AFFECTION-KIND"`
	`  value="PCAffectsPB"/>`

5.9.3. XPath-addressing of ECU-CD using ECU-PD

Because there can be many configurations (using the same definition) and each configuration may use its own short-names for its nodes, addressing configuration-nodes should be done using only the names of the schema-nodes. Otherwise, the addressing would only work for a single configuration.

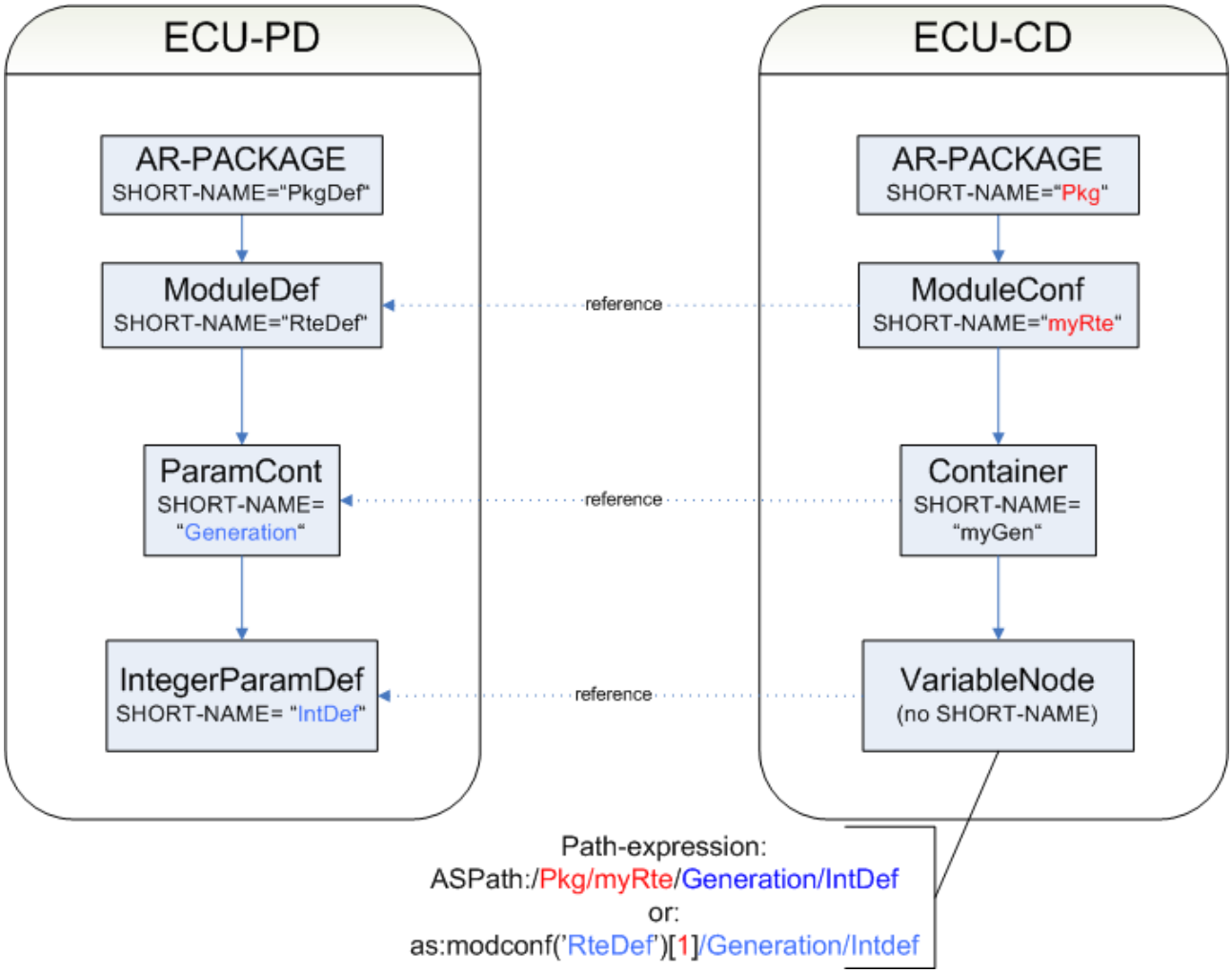


Figure 5.13. XPath-addressing within ECU-CD

Possible XPath-paths to address the ModuleConf:

- ASPath:/Pkg/myRte/Generation/IntDef
- as:modconf('RteDef')[1]/Generation/Intdef

As you can see, the ModuleConf can be addressed using it's configuration-name. However, if the schema-addressing is used instead, it will work for any configuration - regardless of the ModuleConf's name. You only have to define which configuration you want to address (if you have exactly one configuration at a time, this will always be the first configuration).

Note, that the path following "as:modconf()" is an XPath - not a SHORT- NAME-path (otherwise you cannot address the VariableNode). So, if the IntegerParamDef would have a lower or upper multiplicity != 1, there will be a surrounding list and the path would be .../Generation/IntDef/*[1].

The following possibilities to address the VariableNode are only mentioned for completeness:

- as:modconf('RteDef')[name(.)='myRte']/Generation/Intdef
- as:ref(concat(as:path(as:stod('/PkgDef/RteDef')[1]), '/Generation/IntDef'))
- /AUTOSAR/TOP-LEVEL-PACKAGES/*/ELEMENTS/myRte/Generation/Intdef

5.9.3.1. Path in Template

The configurator provides a feature for developers showing the path to address a configuration node using the schema-names. For this, ensure that developer features are enabled. Choose "Window/Preferences..." from the menu.

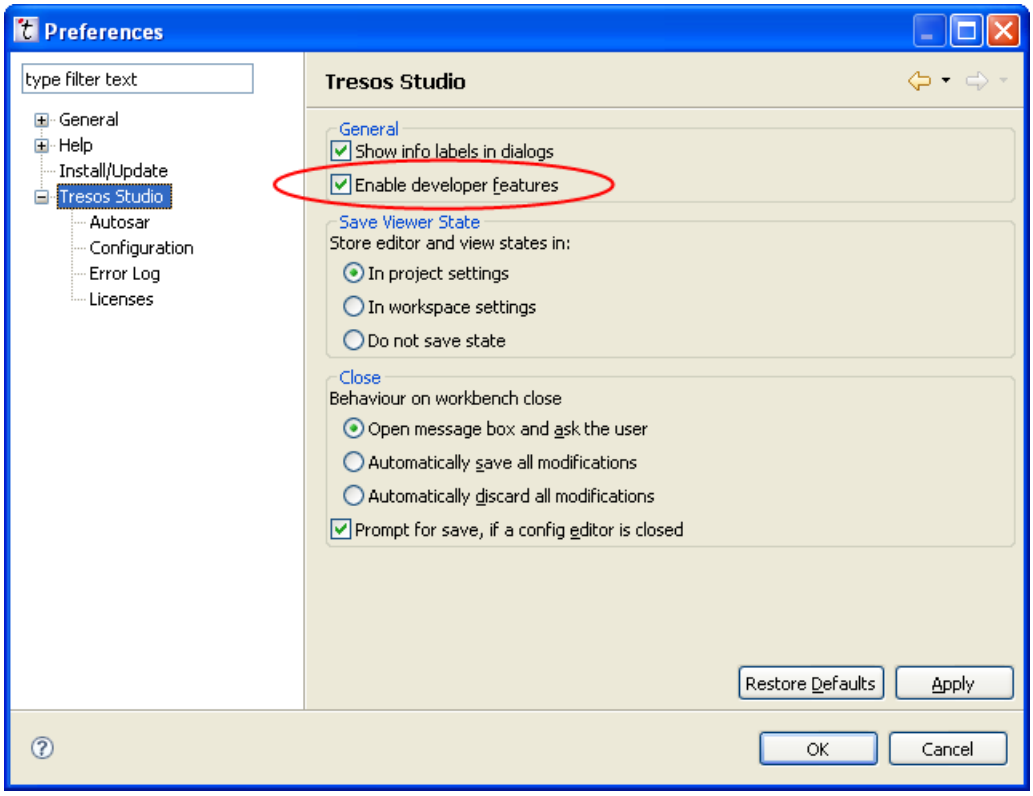


Figure 5.14. Enabling developer features

Now, the "Elt. Info"-view shows the "Path in Template" for the selected configuration-node in the current editor.

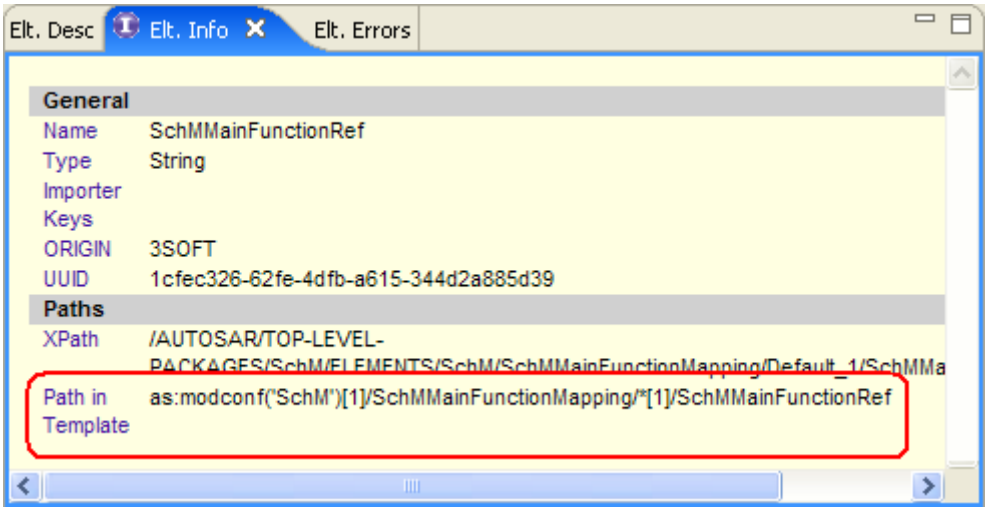


Figure 5.15. Path in template

Note, that you should NOT use this path "as is". If the path contains e.g. "../*[5]/..." it addresses the fifth element within a list. Instead of directly accessing the fifth element, you should loop over all list-elements and use relative path within the loop.

Example:

```
[!LOOP "as:modconf('SchM')[1]/SchMMainFunctionMapping/*"!]
  [!"./SchMMainFunctionRef"!]
[!ENDLOOP!]
```

5.9.4. Top level Structure

The top level node-structure of the datamodel for Autosar-nodes is based on the Autosar-metamodel as depicted below.

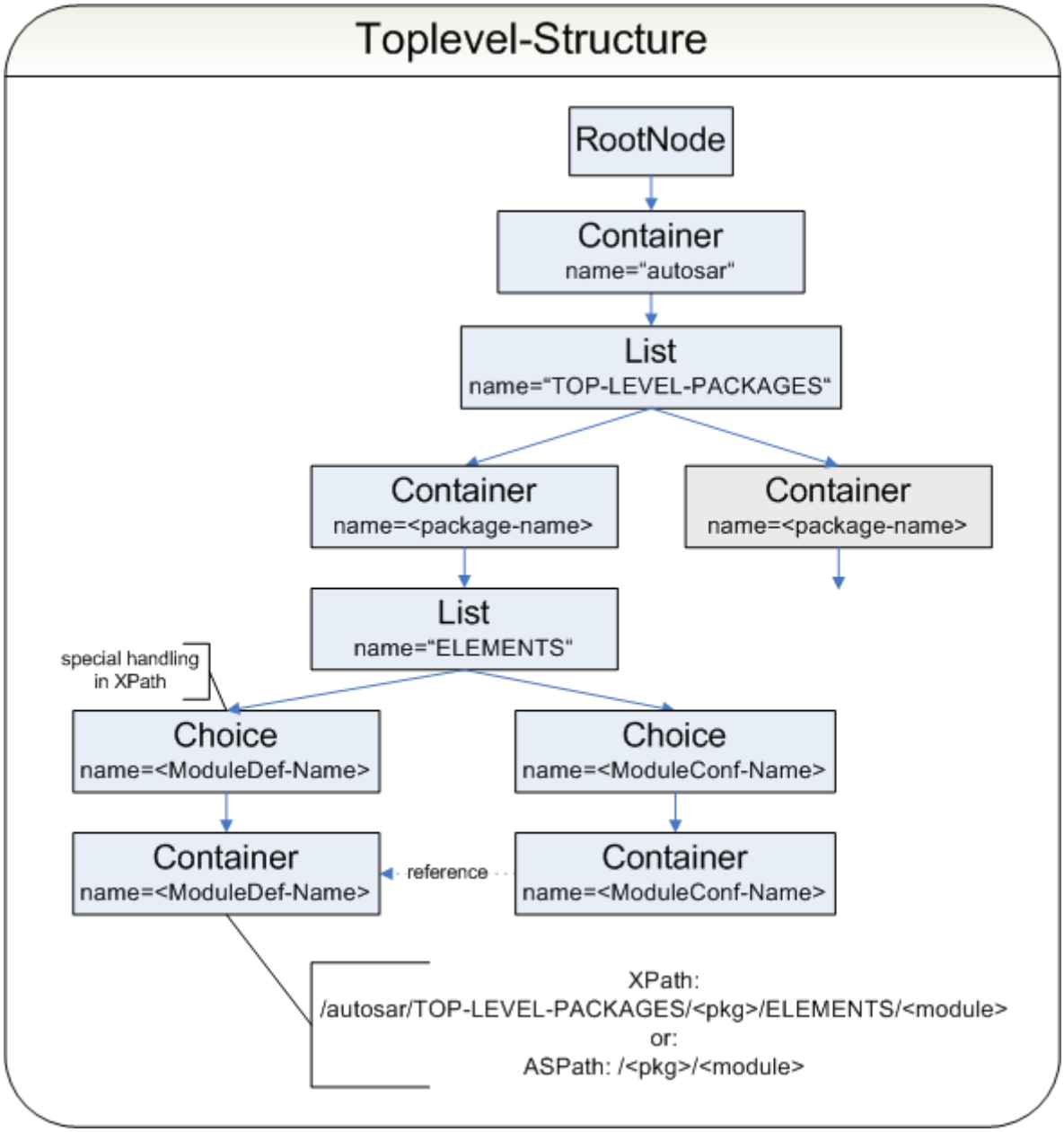


Figure 5.16. Top level structure for Autosar-nodes

XPath	Description
/AUTOSAR/TOP-LEVEL-PACK-AGES/<pkg>	Selects the package with the given name.
/AUTOSAR/TOP-LEVEL-PACKAGES/*/ELEMENTS/<module>	Selects the module-container (not the choice!) with the given name, regardless if it is a definition or a configuration, and regardless of the package .
as:modconf(<module>)	Selects the module-configuration with the given name, regardless in which package it will be found.
ASPath:/<pkg>/<module>	Selects the module-container with the given name in the given package .

Table 5.15. Top level XPath-addressing

5.9.5. List-Nodes

Each node with a lower or upper multiplicity != 1 will be wrapped into a list-node (except optional nodes - see Section 5.9.5.1: Optional elements). The so created list gets the SHORT-NAME of the ECU Parameter Definition-node as its name. This happens both in the schema-tree as well as in the data-tree.

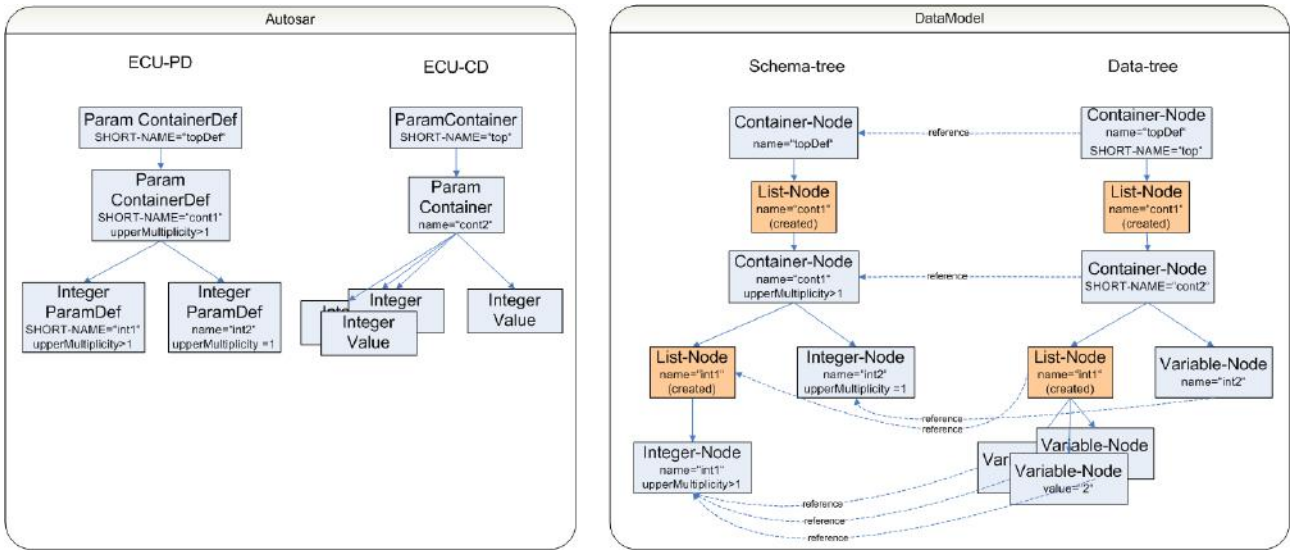


Figure 5.17. List-Representation

XPath (within the data-tree)	Description
topDef/cont1	selects the list "cont2". Note that the selection is done within the data using the schema-name.
topDef/cont1/*[1]	selects the first container within the list (here "cont2"). Note, that the selection is done within the data using the schema-name.
ASPath:top/cont2	selects the container-node "cont2"
topDef/cont1/*[1]/int2	selects the variable-node "int2"
topDef/cont1/*[1]/int1	selects the list "int1"
topDef/cont1/*[1]/int1/*[2]	selects the second variable-node within the list "int1"

Table 5.16. XPath-addressing for Lists

There is no SHORT-NAME-path to a variable-node as this does not have a SHORT-NAME.

5.9.5.1. Optional elements

Autosar is able to define "optional" elements in the ECU Parameter Definition by setting LOWER-MULTIPLICITY == 0 and UPPER-MULTIPLICITY == 1 to an element of the ECU Parameter Definition.

```
<PARAM-CONF-CONTAINER-DEF>
  <SHORT-NAME>MyOptionalContainer</SHORT-NAME>
  <LOWER-MULTIPLICITY>0</LOWER-MULTIPLICITY>
  <UPPER-MULTIPLICITY>1</UPPER-MULTIPLICITY>
</PARAM-CONF-CONTAINER-DEF
```

There are two different possible representations of this construct within the DataModel. The old one (which is no longer recommended) wraps the optional element within a list which can only have one child:

```
<v:lst name="MyOptionalContainer" type=MAP">
  <a:da name="MIN" value="0"/>
  <a:da name="MAX" value="1"/>
  <v:ctr name="MyOptionalContainer"/>
</v:lst>
```

This representation has two major disadvantages:

- The GUI represents each element within its own tab.

- When optional elements are taken over from the StMD to the VSMD as mandatory elements, the Xpath to this element changes. Thus, within a codegenerator- template, two different statements (one for each representation) are needed to safely access this element:

```
[!IF "node:exists(ElementName)"!] ... [!ENDIF!]
[!IF "node:exists(ElementName/*[1])"!] ... [!ENDIF!]
```

The new (and highly recommended) representation simply marks the element as optional (without wrapping it into a list):

```
<v:var name="MyOptionalInteger" type="INTEGER">
  <a:a name="OPTIONAL" value="true"/>
  <a:da name="ENABLE" value="false"/>
</v:var>
```

The xpath to an element does not depend on whether this element is optional or not:

```
[!IF "node:exists(ElementName)"!] ... [!ENDIF!]
```

5.9.5.2. Optional elements within the commandline

To be backward-compatible to existing scripts, the default behavior of the commandline is the "old" list-representation of optional elements. To activate the new recommended representation, the option "MapOptionalAsList" must be set to false. The following example converts a file from the "old" to the new recommended representation:

```
tresos_cmd.bat -DMapOptionalAsList=false legacy convert old.xdm new.xdm
```

5.9.5.3. Optional elements within the GUI

When using the GUI, it depends on the used plugin if optional elements are used. To let a plugin use the optional-feature, its xdm-files must be converted as described above. After that, "old" configuration data files can still be loaded.

5.9.6. Choice-Nodes

Until Autosar 3.0, there was no choice-construct within the ECU Configuration Description. Nonetheless, there is always a choice-data-node within the DataModel. But the Choice-Node within the data-tree cannot be addressed via XPath. Instead, the XPath "topDef/abc" (within the data-tree) returns the chosen container.

For completeness, this choice-data-node refers to the choice-schema-node. The choice-data-node gets the name of the choice-schema-node and the SHORT-NAME of its child as value (the selected container).

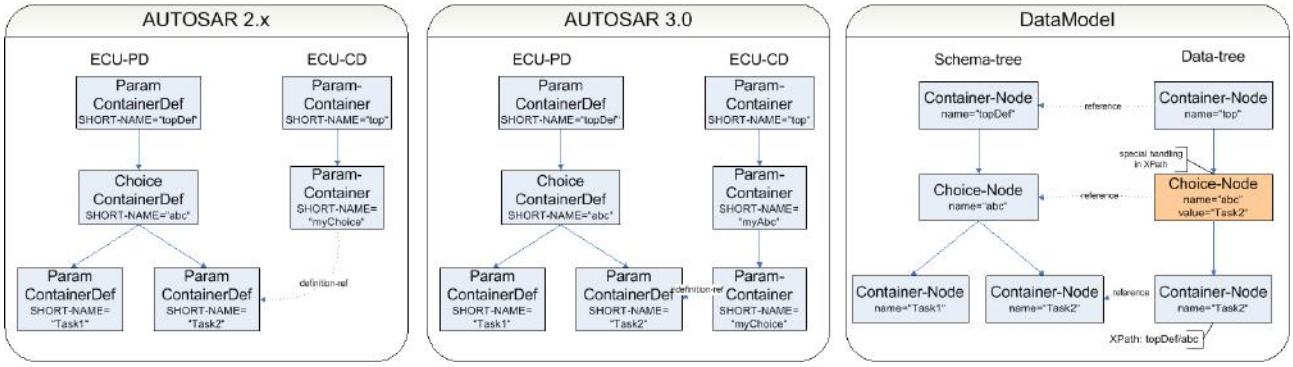


Figure 5.18. Choice-Representation

Note

As described above, choices with a lower or upper multiplicity != 1 will also be wrapped within a list.

The examples in the following table operates on the data-tree shown above.

XPath (within the data-tree)	Description
topDef/abc	selects the container-node "Task2" (the selected container)
ASPath:top/myChoice (Autosar 2.x)	selects the container-node "Task2" (the selected container)
ASPath:top/myAbc/myChoice (Autosar 3.0)	selects the container-node "Task2" (the selected container)

Table 5.17. Choice Examples

5.9.7. References

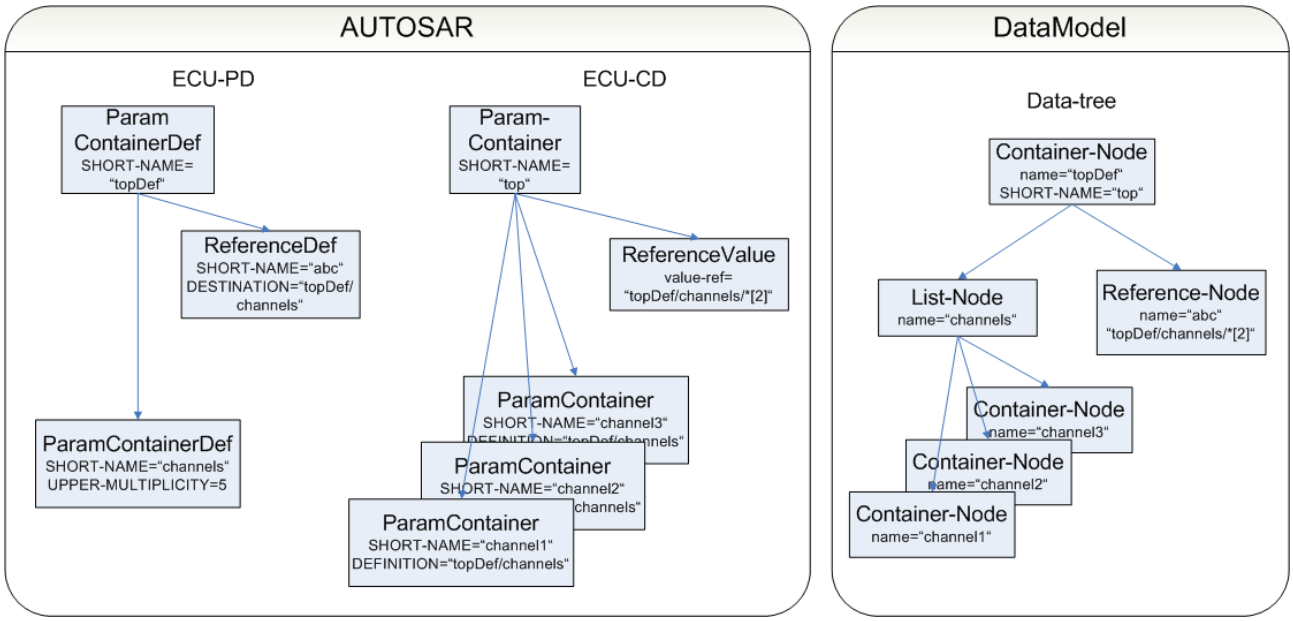


Figure 5.19. Reference-Example

The diagram above shows a part of a data-tree containing a reference-node. The value of the reference-node is the SHORT-NAME-path to the referenced node. The reference-node gets the name from the SHORT-NAME of the ReferenceDef.

XPath (within the data-tree)	Description
topDef/channels/*[2]	selects the container-node 'channel2'
topDef/abc	selects the reference-node
as:ref(topDef/abc) node:ref(topDef/abc)	selects the container-node 'channel2' - the parameter is a reference-node
as:ref('top/channel2')	selects the container-node 'channel2' - the parameter is the SHORT-NAME-path, not the XPath-path!

Table 5.18. Reference-Example

5.10. XPath within the DataModel

XPath is a W3C-Standard and is described in several tutorials available on the web:

W3C-Standard: http://www.w3.org/TR/xpath

http://www.w3schools.com/xpath

http://www.zvon.org/xxl/XPathTutorial/General/examples.html

The basis of how to use XPath within the DataModel is described in this chapter. XPath can be used to navigate within the DataModel, select nodes and retrieve the values of nodes. To retrieve the value of a node, the node must be selected with a corresponding XPath path-expression.

5.10.1. XPath Console

To use the **XPath Console**, you have to enable the preference "Enable developer features" using "Windows/Preferences.../Tresos Studio".

You can open the **XPath Console** from the context-menu of a node within the **Outline view** by choosing "XPath console". The selected node will be used as context-node for all exectued XPath expressions.

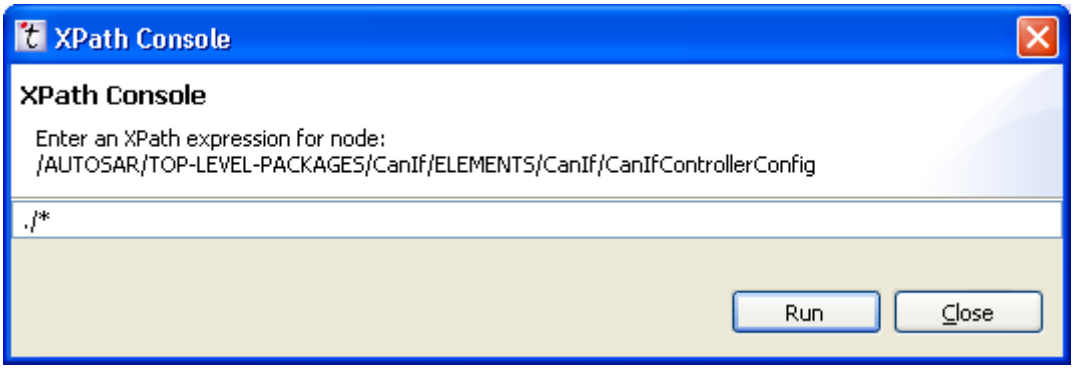


Figure 5.20. XPath Console

Type in any XPath expression and click "Run" to execute it. The result will be shown within the **Search View**.

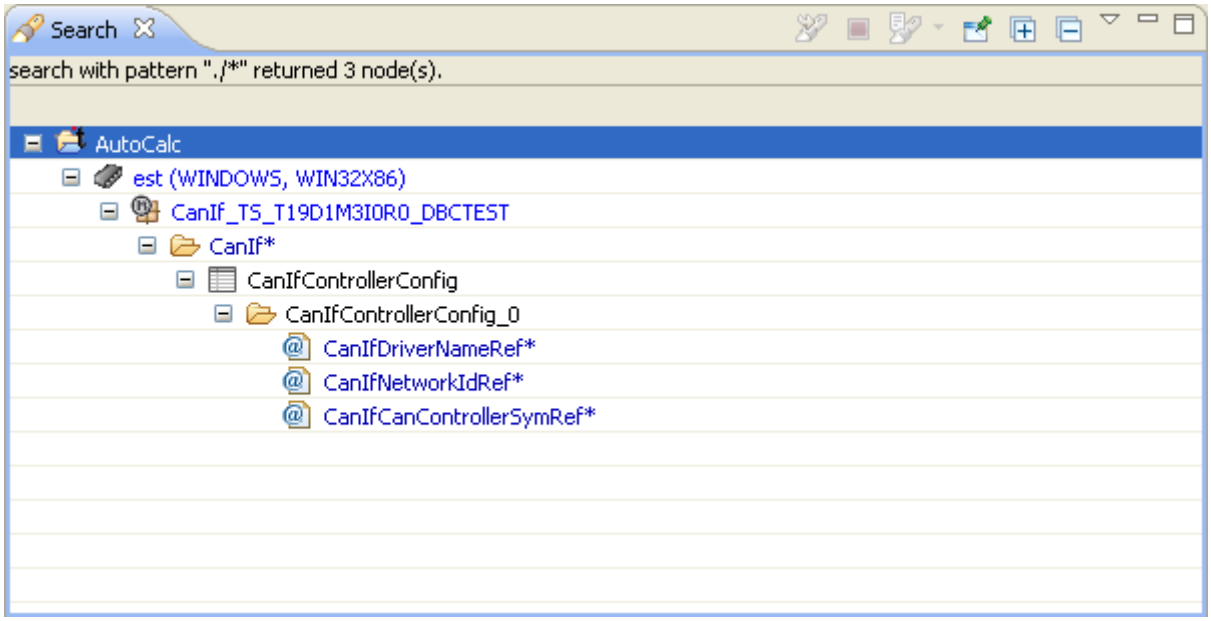


Figure 5.21. XPath Console Result

5.10.2. Conventions

XPath-functions are described in the following form: *return-value* function-name([*parameter[?]] [,parameter[?]][*]*) Example for function 'name' with an optional parameter *node-set* and a *string*-return-value: *string* name(*node-set?*) An *italic* font is used to indicate XPath-data-types (return-value and all parameters). If a parameter is followed by a "?", this parameter is optional - otherwise it is mandatory. A "*" means, that any number of parameters can be given.

5.10.3. Data-Types within XPath

XPath defines the following Data-Types:

1. *node*: a node can be an element, an attribute or simply text
2. *node-set*: an unordered collection of nodes without duplicates.
3. *boolean*: may be true or false
4. *number*: a floating-point number
5. *string*: a sequence of characters
6. *object*: this is no separate data-type but a generalization of the data-types listed above. It is therefore used if any data-type can be returned or given as an argument.

XPath defines rules to automatically convert an expression value to the appropriate type. For example, if a function expects a string as parameter, but got a number, the given number will automatically be converted into a string.

5.10.4. Additional Data-Types for the DataModel

For the use within the DataModel, new data-types are introduced in XPath. For example 'int' is derived from the XPath-data-type 'number' and ensures that there is no floating-point-value.

xpath
: ... a *string* representing a valid XPath- expression

refnode
: ... a *node* representing a reference-node which refers to another node

numlist
: ... a *node-set* of *numbers* or *strings* which represent *numbers*

intlist
: ... a *node-set* of *numbers* without floating-point-value

strlist
: ... a *node-set* of strings

int
: ... a *number* without floating-point-value

hex
: ... a *string* representing a hexadecimal value. The *string* always starts with '0x'

oct
: ... a *string* representing an octal value. The *string* always starts with '\0'

bin
: ... a *string* representing a binary value. The *string* always starts with 'b'

dec
: ... a *string* representing a decimal value.

5.10.5. DataModel-Nodes and their XPath-Value

The following table describes which DataModel-node will return which value.

Node-type	Value in schema	Value in data
Container	empty *string*	the default from schema or empty *string*
Variable	empty *string*	the configured value or the default from the schema
List	empty *string*	the default from schema or empty *string*
Choice	empty *string*	the name of the selected child-node or the default from the schema
Reference	empty *string*	the XPath to the referenced node or the default from the schema

Table 5.19. Values of nodes

5.10.6. Example-Tree

The tree in the picture below is used for all following examples.

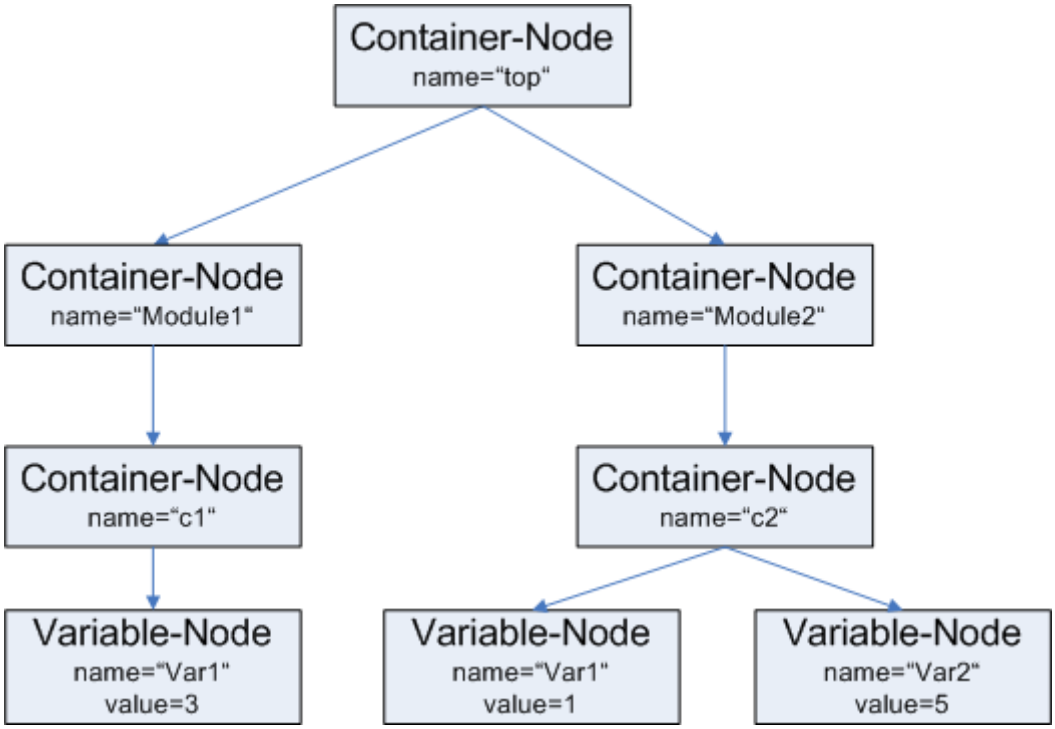


Figure 5.22. Example-Tree

5.10.7. XPath expressions

To select a node, absolute or relative paths can be used with the following expressions:

Expression	Description	Example-XPath	Example-Result
/	selects from the root node	/top/Module1	selects the node 'Module1'
//	selects all nodes in the document from the current node that match the selection no matter where they are	//Var1	selects both(!) nodes with name 'Var1'
.	selects the current node	/top/Module1/.	selects the node 'Module1'
..	selects the parent of the current node	/top/Module1/c1/..	selects the node 'Module1'
*	selects all child-nodes of the current path	/top/*	selects the nodes 'Module1' and 'Module2'
@name	holds the name of the node (for use in predicates - see chapter Section 5.10.8: XPath predicates)	//*[@name='Var1']	selects both(!) nodes with name 'Var1'
@index	holds the index (within it's parents children-list) of the node (for use in predicates - see chapter Section 5.10.8: XPath predicates) note: unlike 'position', the index starts at 0 (zero)	//*[@index>0]	selects node 'Module2' and 'Var2'

Table 5.20. Base XPath-expression

5.10.8. XPath predicates

Predicates are used to find a specific node or a node that contains a specific value. Predicates are always in square brackets.

Syntax	Description	Example	Selected nodes
node/*[position] node-set/*[position]	Selects the child of the node with the given position. Note, that you cannot use a generator-variable here directly. Instead, you have to use the following notation: node/*[position() = $var]	/top/Module2/c2/ *[1]	node 'Var1'
node/*[condition] node-set/*[condition]	selects the child(ren) of the node/node-set which fulfill the given condition	/top/Module2/c2/ *[last()] /top/Module2/ c2/*[last()-1] /top/Module2/ c2/*[position()=2] //*[value>2]	node 'Var2' node 'Var1' node 'Var2' nodes 'Var1' and 'Var2'

Table 5.21. XPath-Predicates

5.10.9. XPath operations

Below is a list of the operators that can be used in XPath expressions:

Operator	Description	Example	Return-Value
|	Combines two node-sets	/top/Module1 | //Module2	nodes 'Module1' and 'Module2'
+	Addition	//Var2 + 7	12.0
-	Subtraction	//Var2 - 1	4.0
*	Multiplication	//Var2 * 2	10.0
div	Division	//Var2 div 1	5.0
=	Equal	//Var2 = 5	true
!=	Not equal	//Var2 != 5	false
<	Less than	//Var2 < 5	false
<=	Less than or equal to	//Var2 <= 5	true
>	Greater than	//Var2 > 5	false
>=	Greater than or equal to	//Var2 >= 5	true
or	or	//Var2 = 7 or /top/Module1/c1/Var1 = 3	true
and	and	//Var2 = 7 and /top/Module1/c1/Var1 = 3	false
mod	Modulus (div remainder)	//Var2 mod 2	1.0

Table 5.22. XPath-operators

5.10.10. XPath functions

XPath contains a small set of build-in-functions. The most useful ones are listed in the following tables.

5.10.10.1. Node-Set functions

Function	Description	Example	Return-value
number last()	returns the position-number of the last child of the context-node	//c2/*[last()]	node 'Var2'
number position()	returns the position-*number* of the context-node	//c2/*[position() > 0]	all child-nodes of 'c2': 'Var1' and 'Var2'
number count(*node-set*)	returns the *number* of nodes within the given *node-set* Hint: use *num:i* to get integer-values.	count(//*) count(//Module2/*)	8.0 1.0
string name(*node-set*)	returns the name of the first context-node. If the node has no name, the position of the node within it's parent's child-list containing two underlines in front of the node's position number is returned, e.g.: '__2'.	name(//c2/*)	'Var1'

Table 5.23. Node-Set Functions

5.10.10.2. String-functions

Function	Description	Example	Return-value
string string(*object*)	converts the given *object* into a *string*	string(5)	'5'
string concat(*string*, *string**)	concatenates the given *strings*	concat('a', 'b', 'c')	'abc'
boolean starts-with(*string*, *string*)	returns true, if the first argument starts with the second argument	starts-with('abctest', 'abc')	true
boolean contains(*string*, *string*)	returns true, if the first argument contains the second argument	contains('abcde', 'bc')	true
string substring-before(*string*, *string*)	returns the substring of the first argument string that precedes the first occurrence of the second argument string with the first argument string, or the empty string if the first argument *string* does not contain the second argument *string*	substring-before('1999/04/01', '/')	'1999'
string substring-after(*string*, *string*)	returns the sub*string* of the first argument *string* that follows the first occurrence of the second argument *string* in the first argument *string*, or the empty string if the first argument *string* does not contain the second argument string	substring-after('1999/04/01', '/')	'04/01'

Table 5.24. String-Functions

string substring(*string*, *number*, *number*?)	returns the sub*string* of the first argument starting at the position specified in the second argument with the length specified in the third argument. If the third argument is not specified, the substring starts at the position specified in the second argument and continues until the end of the *string*	substring('12345', 2, 3) substring('12345', 2)	'234' '2345'
number string-length(*string*?)	returns the *number* of characters in the *string*. If the argument is not specified, the length of the value of the context-node will be returned.	string-length('12345')	5.0
string normalize-space(*string*?)	returns the argument *string* with whitespace normalized by stripping leading and trailing whitespace and replacing sequences of whitespace characters by a single space. If the argument is omitted, it defaults to the *string*-value of the context-*node*.	normalize-space(' 12 345 ')	'12 345'
string translate(*string*, *string*, *string*)	returns the first argument *string* with occurrences of characters in the second argument string replaced by the character at the corresponding position in the third argument *string*	translate('bar', 'abc', 'ABC') translate('-aaa---', 'abc-', 'ABC')	'BAr' 'AAA'

Table 5.25. String-functions

5.10.10.3. Boolean-functions

Function	Description	Example	Return-value
boolean boolean(*object*)	converts it's argument to a *boolean* as follows: a *number* is true if it is non-zero a *node-set* is true if it is non-empty a *string* is true if it's length is non-zero	boolean(5) boolean(0) boolean('')	true true false false
boolean not(*boolean*)	returns true if it's argument is false, and false otherwise	not(true())	false
boolean true()	returns true	true()	true
boolean false()	returns false	false()	false
boolean lang(*string*)	not supported		

Table 5.26. Boolean-functions

5.10.10.4. Number-functions

Function	Description	Example	Return-value
number number(*object*?)	converts it's argument to a *number* as follows: - a *string* that consists of optional whitespace followed by an optional minus sign, followed by a *number*, followed by whitespace is converted into a number that is nearest to the mathematical value represented by the string; - *boolean* true is converted to 1 - *boolean* false is converted to 0 - a *node-set* is converted to a *string* in the same way a call to the string-function would convert it. It is then converted in the same way as a *string* argument	number('5')	5.0
number sum(*node-set*)	returns the sum of each *node* in the argument *node-set*	sum(//*[position() > 0])	9.0
number floor(*number*)	returns the largest (closest to positive infinity) *number* that is not greater than the argument and that is an *integer*	floor(5.3)	5.0
number ceiling(*number*)	returns the smallest (closest to negative infinity) *number* that is not less than the argument and that is an *integer*	ceiling(5.3)	6.0
number round(*number*)	returns the *number* that is closest to the argument and that is an *integer*	round(1.1) round(1.5) round(1.9)	1.0 2.0 2.0

Table 5.27. Number-functions

5.10.10.5. Additional functions provided by the XPath Engine

The following functions are added to provide additional functionality especially for accessing the data-structure within the DataModel. *Hint:* The default prefix for the function namespace is fn: and is therefore omitted in the previous tables. Actually it does not matter, whether you write 'fn:round(3.5)' or 'round(3.5)'. In the following table, other function-namespaces are used and thus cannot be omitted.

5.10.10.5.1. Node functions

Function	Description	Example	Return-value
node node:current()	returns the current *node*	node:current()	current node
node node:islast()	returns true if the context-node is the last element in its surrounding list. Note: This does NOT mean, that it will return true for the last node of a LOOP or SELECT!	node:islast()	true
string node:name(*node*)	returns the name of the given *node*	node:name(//Var2)	'Var2'
node node:ref(*refnode*) *node* node:ref(*xpath*)	returns the *node* which is referenced by the given *node* (or target-*node* of the *xpath*-path)	node:ref('/myNode') node:ref(/myReference)	The node "myNode". The target-node of node "myReference".
nodeset node:refs(*nodeset*) *nodeset* node:refs(*xpath*)	returns the *nodes* which are referenced by the given *nodes* (or target-*nodes* of the *xpath*-path). Note that an *xpath*-path (parameter surrounded by apostrophes) never returns multiple nodes. To give in multiple nodes, an *xpath*-expression must be used (not surrounded by apostrophes).	node:refs(/top/Module2/c2/*) node:refs('/top/Module2/c2/*')	First example returns Var1 and Var2 (because its an *xpath*-expression). Second example only returns Var1 (because its an *xpath*-path).
node:value(*node*)	returns the value of the *node*	node:value(//Var2)	5
node:value(*String*)	returns the value of the *node* represented by its path	node:value(concat("/top/Module2", "/c2/Var2"))	5
node node:dtos(*dataPath*) *node* node:dtos(*dataNode*)	returns the schema-*node* which is referred to by the given data-node or the data-node described by the given path. "dtos" stands for DataTOSchema.	as:dtos('/data1') as:dtos(/data1)	the corresponding schema-node
nodeset as:stod(*schemaPath*) *nodeset* as:stod(*schemaNode*)	returns the data-*node*s which are referencing the given schema-node or the schema-node described by the given path. "stod" stands for SchemaTOData.	as:stod('/schema1') as:stod(/schema1)	all data-nodes for the given schema-node
string node:path(*node*)	returns the XPath of the *node*	node:path(/Var)	"/Var"

Table 5.28. Node functions

nodeset node:paths(*node-set*)	returns the XPaths of the *node(s)*	node:paths(//Var)	all XPaths for all nodes named "Var"
nodeSet node:order(*nodeSet*[, condition1[, condition2]])	Orders the given nodeSet. The conditions (which must be xpath-expressions) will be used to order the nodes. Within the condition, you can access the current node of the nodeSet. If no condition is given, the nodes are ordered by their name.	node:order(//*, 'node:value(.)', 'node:name(.)')	Orders all nodes by their values. Nodes with equal values will be ordered by their name.
boolean node:exists(*node*)	Checks if the given node exists (without generating an error).	node:exists(nonode), not(node:exists(nonode))	false, true
boolean node:refexists(*node*)	returns true, if the given reference-node and its referenced node exists (without generating an error).	node:refexists(refnode)	true, if the given node references an existing node
boolean node:refvalid(*node*)	returns true, if the given reference-node and its referenced node exists and if the referenced node is allowed to be referenced.	node:refvalid(refnode)	true, if the given node references an existing valid node
boolean node:empty(*node*)	returns true, if the given node does not exist or if its value is empty (without generating an error) - false otherwise.	node:empty(*node*)	true, if the given node does not exist or its value is empty
node-set node:difference(*node-set1, node-set2*)	Removes the "remaining quantity" of the node-set1 (returns the node-set1 not containing any nodes of the node-set2).	node:difference(//Var, /Os/Var)	all "Var"-nodes but not "/Os/Var".
boolean node:isconsecutive(*node-set*, base)	Returns true, if the given node-set is a consecutive list of integer-nodes based on the given base. For this, the nodes must not be ordered (e.g. 0, 2, 1 is accepted as zero-based consecutive list).	node:isconsecutive(MyList/*, 0)	true, if all given nodes are zero-based consecutive

Table 5.29. Node functions

5.10.10.5.2. Text functions

Function	Description	Example	Return-value
numlist text:range(*number*, *number*)	Creates a *list* of *number*s over which an XPath expression can iterate, for example. The next *number* is generated by adding or subtracting 1. - param1: The first *number* to generate - param2: The last *number* to generate	text:range(2, 4)	(2.0, 3.0, 4.0)
numlist text:range(*number*, *number*, *number*)	Creates a *list* of *number* over which an XPath expression can iterate, for example. The next *number* is generated by adding or subtracting step. - param1: The first *number* to generate param2: The last *number* generated is not higher (or smaller if end < start) than end - param3: The incremental steps	text:range(2, 8, 2)	(2.0, 4.0, 6.0, 8.0)
strlist text:split(*object*)	Splits the *string*-representation of the given *object*(s) into a *set of nodes* at space boundaries. - param: The data to split up	text:split('1 2 3')	('1', '2', '3')
strlist text:split(*object*,*string*)	Splits a *string* into a set of *nodes* with a given delimiter. - param1: The data to split up - param2: The element delimiter	text:split('1,2,3', ',')	('1', '2', '3')

Table 5.30. Text functions

string text:join(*node-set*)	Joins the *string* representation of a list of *nodes* with spaces to a single *string*. - returns a *string* with the *node* value delimited by a space	text:join(//Var1)	'3 1'
string text:join(*node-set*,*string*)	Joins the *string* representation of a list of *nodes* with a given delimiter to a single *string*. - param1: the *nodes* to join - param2: the delimiter that should be put between the *node* values - returns a *string* with the *node* value delimited by a space	text:join(//Var1, ',')	'3,1'
boolean text:match(*string*, *string*)	Performs a regular expression pattern match on a *string*. - param1: The text to search - param2: The regexp pattern to use - returns true if the pattern matches	text:match('123', `'^[0-9]*')`	true
strlist text:grep(*node-set*,*string*)	Performs a regular expression pattern-match on the *string*-representation of a list of nodes returning the string-representations of all *nodes* that match. - param1: The *nodes* to match - param2: The regexp pattern to use - returns the *string*-representation of all nodes that did match the pattern	text:grep(text:split('1 2 3'), `'^[2-9]*')`	('2', '3')

Table 5.31. Text functions

string text:replace(*string*, *string*, *string*)	Performs a regular expression replace operation. The first occurrence matching pattern is replaced by the replacement string - param1: The input text - param2: The search pattern. The part of the text that matches is replaced - param3: The replacement *string* - returns the replaced text	text:replace('12345', '3', '0')	'12045'
string text:replaceAll(*string*, *string*, *string*)	Performs a regular expression replace operation. All occurrences matching pattern are replaced by the replacement string - param1: The input text - param2: The search pattern. The part of the text that matches is replaced - param3: The replacement *string* para - returns the replaced text	text:replaceAll('12321', '1', '0')	'02320'
boolean text:uniq(*node-set*,*string*)	Tests if the value is unique inside the *node-set*. - param1: The list to search - param2: The value to search inside the list. - returns true if the value occurs at most once inside the list.	text:uniq(text:split('1 2 1'), '1')	false
node-set text:difference(*node-set1, node-set2*)	Returnes the "remaining quantity" of node-set1 (returns the node-set1 not containing any string of the node-set2).	text:difference(text:split('1 2 3 4'), text:split('1 3'))	('2', '4')
strlist text:concat(*node-set*, *node-set*)	concatenates the *string*-representation of the given *objects*. The first element of the returned *node-set* is the concatenation of the first elements of both given *node-sets*. If one of the two *node-sets* is shorter than the other, it's last element is used for the open calculations.	text:concat(text:split('1 2 3'), text:split('4 5'))	('14', '25', '35')

string text:toupper(*string*)	converts all of the characters in the given string to upper case	text:toupper('small')	'SMALL'
string text:tolower(*string*)	converts all of the characters in the given string to lower case	text:tolower('BIG')	'big'
string text:order(*node-set*)	sorts the *string*-representation of the given *objects* in ascending order and returns the sorted list as *node-set*.	text:order(text:split('zoo foo boo'))	('boo','foo','zoo')
string text:order(*node-set*, *direction*)	sorts the *string*-representation of the given *objects* in the order given by *direction* and returns the sorted list as *node-set*. If *direction* is < 0, the order is descending, >= 0 means ascending order	text:order(text:split('zoo foo boo', -1))	('zoo','foo','boo')
string text:order(*node-set*, *string*, *number*)	Sorts the *string*-representation of the given *objects* in ascending order and returns the sorted list as *node-set*. A pattern and a group index are used to compute the portion of the strings which should be considered for the sort condition. - param1: The list to sort - param2: A regular expression which matches a substring of the strings to sort. The regexp may contain groupings (round brackets) - param3: The group index of the regular expression which should be used for the sort condition; index 0 is the whole regular expression match, 1 is the first group, ... - returns the sorted list	text:order(text:split('zoo-b-b goo-zz-a foo-a-c boo-z-d'), '.*?-([^-]+)', 1)	('foo-a-c', 'zoo-b-b', 'boo-z-d', 'goo-zz-a')

string text:order(*node-set*, *string*, *number*, *direction*)	Sorts the *string*-representation of the given *objects* in the given order and returns the sorted list as *node-set*. A pattern and a group index are used to compute the portion of the strings which should be considered for the sort condition. - param1: The list to sort - param2: A regular expression which matches a substring of the strings to sort. The regexp may contain groupings (round brackets) - param3: The group index of the regular expression which should be used for the sort condition; index 0 is the whole regular expression match, 1 is the first group, ... - param4: If the value is < 0, the sort order is descending, if it is >= 0, the order is ascending - returns the sorted list	text:order(text:split('zoo-b-b goo-zz-a foo-a-c boo-z-d'), '.*?-([^-]+)', 1, -1)	('goo-zz-a', 'boo-z-d', 'zoo-b-b', 'foo-a-c')

Table 5.32. Text functions

5.10.10.5.3. Number functions

Function	Description	Example	Return-value
numlist num:add(*numlist*, *numlist*)	adds the values of the two lists. If one of the two lists is shorter, then it's last element is used for the open calculations.	num:add(text:split('1 2 3'), text:split('4 5'))	(5.0, 7.0, 8.0)
numlist num:div(*numlist*, *numlist*)	divides the values of the two lists. If one of the two lists is shorter, then it's last element is used for the open calculations.	num:div(text:split('8 9 6'), text:split('2 3'))	(4.0, 3.0, 2.0)
boolean num:isnumber(*object*)	returns true, if the given *object* is a or can be converted to a *number*	num:isnumber('3')	true
numlist num:mod(*numlist*, *numlist*)	computes the modulus for the values of the two lists. If one of the two lists is shorter, then it's last element is used for the open calculations.	num:mod(text:split('5 6 7'), text:split('2 3 4'))	(1.0, 0.0, 3.0)
numlist num:mul(*numlist*, *numlist*)	multiplies the values of the two lists. If one of the two lists is shorter, then it's last element is used for the open calculations.	num:mul(text:split('1 2 3') text:split('3 2 3'))	(3.0, 4.0, 9.0)
numlist num:negate(*numlist*)	negates all values of the list	num:negate(text:split('-1 0 3'))	(1.0, 0.0, -3.0)
intlist num:sub(*numlist*)	subtracts the values of the two lists. If one of the two lists is shorter, then it's last element is used for the open calculations.	num:sub(text:split('4 5 6') text:split('3 2 1'))	(1.0, 3.0, 5.0)
int num:i(*numlist*)	Returns an *integer* value from any *number* (does a floor).	num:i('5.3')	5
intlist num:order(*numlist*)	sorts the given numbers in ascending order and returns the result.	num:order(text:split('-5.5 3 6 1 5 -6 0 3.5 -5.3'))	(-6, -5.5, -5.3, 0, 1, 3, 3.5, 5, 6)
intlist num:order(*numlist*, *direction*)	Sorts the given numbers in the given order and returns the result. If *direction* is < 0, the order is descending, if *direction* is >= 0, the order is ascending.	num:order(text:split('-5.5 3 6 1 5 -6 0 3.5 -5.3', -1))	(6,5, 3.5, 3, 1, 0, -5.3, -5.5, -6)

Table 5.33. Number functions

int num:hextoint(*string*)	Converts the given *string* into an *integer*. The string represents a hexa-decimal number ('0x<hex-number>' or '<hex-number>'). - param1: String representing a hexa-decimal number - returns an *integer* representation of the number - handles 56-bit signed values	num:hextoint('0xA') num:hextoint('0xfb')	10 251
int num:octtoint(*string*)	Converts the given *string* into an *integer*. The string represents an octal number ('\0<oct-number>' or '<oct-number>'). - param1: String representing an octal number -returns an *integer* representation of the number - handles 56-bit signed values	num:octtoint('\010')	8
int num:bintoint(*string*)	Converts the given *string* into an *integer*. The string represents a binary number ('b<bin-number>' or '<bin-number>'). - param1: String representing a binary number - returns an *integer* representation of the number - handles 56-bit signed values	num:bintoint('b11')	3
int num:dectoint(*string*)	Converts the given *string* into an *integer*. The string represents a decimal number. - param1: String representing a decimal number - returns an *integer* representation of the number - handles 56-bit signed values	num:dectoint('10')	10

int num:radixtoint(*string*)	Converts the given *string* representing a number with arbitrary radix into an *integer* (uses the corresponding prefix '0x<hex-number>', '\0<oct-number>', 'b<bin-number>' as the value type indication). - param1: String representing a number to convert - returns an *integer* representation of the *number* - handles 56-bit signed values	num:radixtoint('0xA') num:radixtoint('b11')	10 3
string num:inttobin(*int*)	Starting with *b*, num:inttobin returns a string representation of the given integer in the binary system. - param1: Integer to convert - returns a string representing the given integer in binary system - handles 56-bit signed values	num:inttobin(3)	'b11'
string num:inttobin(*int*, *int*)	Starting with *b*, num:inttobin returns a string representation of the given integer in the binary system. The second *integer* argument specifies a fix digit length of the resulting string (without b). Either leading zeros are inserted or the result may be trimmed to fit this length. - param1: Integer to convert - param2: Digit length of the resulting string (without b) - returns a string representing the given integer in binary system. - handles 56-bit signed values	num:inttobin(3, 4)	'b0011'
string num:inttohex(*int*)	Starting with *0x*, num:inttohex returns a string representation of the given integer in the hexa-decimal system. - param1: Integer to convert - returns a string representing the given integer in hexa-decimal system - handles 56-bit signed values	num:inttohex(15) num:inttohex(254)	'0xf' '0xfe'

string num:inttohex(*int*, *int*)	Starting with *0x*, num:inttohex returns a string representation of the given integer in the hexa-decimal system. The second *integer* argument specifies a fix digit length of the resulting string (without 0x). Either leading zeros are inserted or the result may be trimmed to fit this length. - param1: Integer to convert - param2: Digit length of the resulting string (without 0x) - returns a string representing the given integer in hexa-decimal system. - handles 56-bit signed values	num:inttohex(15, 2) num:inttohex(254, 1)	'0x0f' '0xe'
string num:inttooct(*int*)	Starting with *\0*, num:inttooct returns a string representation of the given integer in the octal system. - param1: Integer to convert - returns a string representing the given integer in octal system - handles 56-bit signed values	num:inttooct(15)	'\017'
string num:inttooct(*int*, *int*)	Starting with *\0*, num:inttooct returns a string representation of the given integer in the octal system. The second *integer* argument specifies a fix digit length of the resulting string (without \0). Either leading zeros are inserted or the result may be trimmed to fit this length. - param1: Integer to convert - param2: Digit length of the resulting string (without \0) - returns a string representing the given integer in octal system. - handles 56-bit signed values	num:inttooct(15, 4)	'\00017'

Table 5.34. Number conversions

5.10.10.5.4. Bit functions

Function	Description	Example	Return-value
int bit:and(*int*,*int*)	Does a bitwise and of two *integer* values. - var1: First *integer* value - var2: Second *integer* value - returns anded value of the two *integers* - handles 56-bit signed values	bit:and(5, 3)	1
int bit:or(*int*,*int*)	Does a bitwise or of two *integer* values. - var1: First *integer* value - var2: Second *integer* value - returns or-ed value of the two *integers* - handles 56-bit signed values	bit:or(5, 3)	7
int bit:xor(*int*,*int*)	Does a bitwise xor of two *integer* values. - var1: First *integer* value - var2: Second *integer* value - returns xor-ed value of the two *integers* - handles 56-bit signed values	bit:xor(5,3)	6

Table 5.35. Bit functions

int bit:not(*int*)	Does a bitwise not on an *integer* value. - var1: *integer* value - returns inverted value of the *integer* - handles 56-bit signed values	bit:and(bit:not(3), 7)	4
int bit:shl(*int*,*int*)	Does a bitwise shift left on an *integer* value. - param1: *integer* value - param2: *number* of bits to shift left - returns the shifted value - handles 56-bit signed values	bit:shl(5,1)	10
int bit:shr(*int*,*int*)	Does a bitwise shift right on an *integer* value. - param1: *integer* value - param2: *number* of bits to shift right - returns the shifted value - handles 56-bit signed values	bit:shr(10,1)	5
int bit:bitclear(*int*, *int*)	clears the <param2>th bit of the given param1. The first bit is bit nr. 0. - handles 56-bit signed values	bit:bitclear(7,2)	3
int bit:bitset(*int*, *int*)	sets the <param2>th bit of the given param1. The first bit is bit nr. 0. - handles 56-bit signed values	bit:bitset(3,2)	7
boolean bit:getbit(*int*, *int*)	returns true, if the <param2>th bit is set within param1. Otherwise false is returned. The first bit is bit nr. 0. - handles 56-bit signed values	bit:getbit(8,1)	false

Table 5.36. Bit functions

5.10.10.5.5. Variable functions

Function	Description	Example	Return-value
boolean var:defined('varName')	Checks if the variable with the given name is defined (assigned to a value) without generating an error.	var:defined('myVar') not(var:defined('myVar'))	true, if the var is defined, false otherwise
boolean var:set('varName', 'value')	Defines an xpath variable and sets its value. If the variable already exists, its value is overwritten.	var:set('myVar','myValue')	true, if the variable could be set, false otherwise

Table 5.37. Variable functions

5.10.10.5.6. License functions

provides methods for querying the license information.

Function	Description	Example	Return-value
boolean lic:feature('feature')	returns true, if the requested feature is unlocked by the base-license (`LICENSE_ID == tresosECU`).	lic:feature('GUI') not(lic:feature('GUI'))	true, if the base-license unlocks feature "GUI" is installed
boolean lic:feature('moduleId', 'feature')	returns true, if the license for the given module unlocks the given feature.	lic:feature('Can_TS_T16D4M20R0', 'GUI') not(lic:feature('Can_TS_T16D4M20R0', 'GUI'))	true, if the license for module 'Can_TS_T16D4M20R0' unlocks feature GUI.

Table 5.38. License functions

5.10.10.6. Additional XPath-Functions for Autosar

Each path-expression prefixed with "ASPath:" will be interpreted SHORT-NAME-path.

So, for example, if "/MyPackage/Can" is a valid SHORT-NAME-path:

```
as:ref('/MyPackage/CAN')
  is equivalent to
ASPath:/MyPackage/Can
```

```
  is equivalent to
/AUTOSAR/TOP-LEVEL-PACKAGES/MyPackage/ELEMENTS/Can
```

ASPath:/MyPackage/Can can be used anywhere as xpath-expression and will always be interpreted as SHORT-NAME-path. The conversion from an XPath-path to a SHORT-NAME-path can be done with the function as:path as described in the table below.

Nodes without SHORT-NAME (e.g. lists, choices and variable- nodes) cannot be addressed when using the SHORT-NAME-path.

To provide convenient access to the Autosar node-structure, additional functions have been introduced to XPath:

Function	Description	Example	Return-value
node as:ref(*refnode*)	returns the *node*, which is referred to by the given *refnode* *refnode*: may be a 'Reference', 'Choice Reference' or a 'Symbolic Name Reference'	see Section 5.9.7: References	
node as:ref(*string*)	returns the *node*, which is refereed to by the given SHORT-NAME-path	as:ref('/myPkg/myModule')	the module "myModule" within package "myPkg"
node as:dtos(*dataPath*) *node* as:dtos(*dataNode*)	returns the schema-*node* which is referred to by the given data-node or the data-node described by the given path (which must be a valid SHORT-NAME-path). "dtos" stands for DataTOSchema.	as:dtos('/AUTOSAR/ ModuleConf') as:dtos(/path/to/dataNode)	the corresponding module-definition
nodeset as:stod(*schemaPath*) *nodeset* as:stod(*schemaNode*)	returns the data-*node*s which are referencing the given schema-node or the schema-node described by the given path (which must be a valid SHORT-NAME-path). "stod" stands for SchemaTOData.	as:stod('/AUTOSAR/ ModuleDef') as:stod(/path/to/schemaNode)	all module-descriptions whose definition-path are pointing to the current schema-node
string as:path(*node*)	the SHORT-NAME-path for the given node	as:path(/top/Module2)	the SHORT-NAME-path for Module2
nodeset as:modconf(*modDefName*)	returns the module-configurations which referred to a module-definition with the given name	as:modconf('Rte')	all module-configs for the module-definition Rte
string as:uuid()	returns a new random uuid.	as:uuid()	A random uuid e.g.: 05563-1234-343434-34322

Table 5.39. Additional XPath-Functions for Autosar

5.10.11. Pitfalls when using XPath

This chapter contains common pitfalls when using XPath in conjunction with EB tresos Studio.

5.10.11.1. Access to non-existing nodes in comparisons

Usually, an exception is thrown when you access a non-existing node. However, when you compare a node with another value, no exception is thrown, if there is no node. Instead, an empty list is used internally.

As a consequence, the result of a comparison with a non-existing node is always the same:

```
./NonExistingNode = 'STANDARD' // always returns false
./NonExistingNode != 'STANDARD' // always returns true
```

This may cause unexpected behavior in code templates:

```
[!IF "./NonExistingNode = 'STANDARD'"!]
  // never executed
[!ELSEIF "./NonExistingNode = 'MIXED'"!]
  // never executed
[!ELSE!]
  // always executed
[!ENDIF!]
```

5.10.11.2. Using node values and variables

XPath always interprets the value of nodes and variables as a string. This is especially not intuitive for integer, float and boolean values.

These expressions are all wrong. Do *not* use the following expressions:

```
$var1 = true()     // wrong!
./myBool = true() // wrong!
./myInt = 3        // wrong!
node:exists(.) = true() // wrong!
```

Instead, use the following expressions:

```
$var1    = 'true'
./myBool = 'true'
./myInt  = '3'
```

```
node:exists(.) = 'true'
```

5.10.11.3. Values that represent a logical false in XPath

There are several values, which represent a logical false in XPath. Therefore you may not compare them with the string *false* or the empty string '':

```
node:exists(.) = 'false'     // wrong
node:exists(.) = not('true') // wrong
node:exists(.) != 'true'     // ok
```

5.11. GUI Annotation in XDM

The DataModel and therefore the XDM format supports adding GUI Annotations the parameter definition to allow a more finely granulated control about how the configuration is rendered in a generic configuration editor. This chapter introduces all these constructs and describe how they are formulated in XDM.

5.11.1. Generic Configuration Editor Layout

This chapter details how a configuration model is laid out in the GUI. It assumes that an Autosar parameter definition already forms the basis of the configuration.

The generic configuration editor reads a parameter definition and creates a respective GUI.

The generic configuration editor is composed of two parts:

1. The Editor itself
2. The Node Outline

The editor displays pages of which only one is visible at a time. The generic configurator creates a page for the `MODULE-DEF` element and one for each container that is part of a list (`UPPER-MULTIPLICITY` or `LOWER_MULTIPLICITY` != 1).

A page has a title and an entry that allows editing the name (`SHORT-NAME` of the underlying container). If the page represents the module definition, the name is not editable.

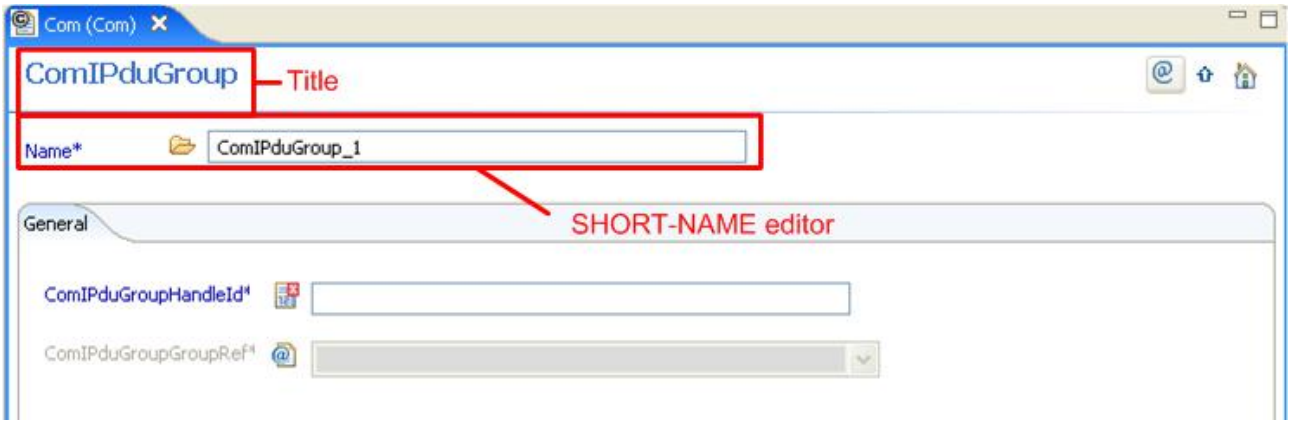


Figure 5.23. Top of page

The page contains editing elements for all configuration parameters, sub-containers and choices beneath the container represented by the page.

The page content is grouped into tabs. A `General` tab is created if there are parameters with `UPPER-MULTIPLICITY` or `LOWER_MULTIPLICITY` != 1 (in this case the parameter, container or choice is represented by a list in the DataModel).

All list elements (`UPPER-MULTIPLICITY` or `LOWER_MULTIPLICITY` != 1) as well as instance references are stored on an own page which is labeled with the name of the element.

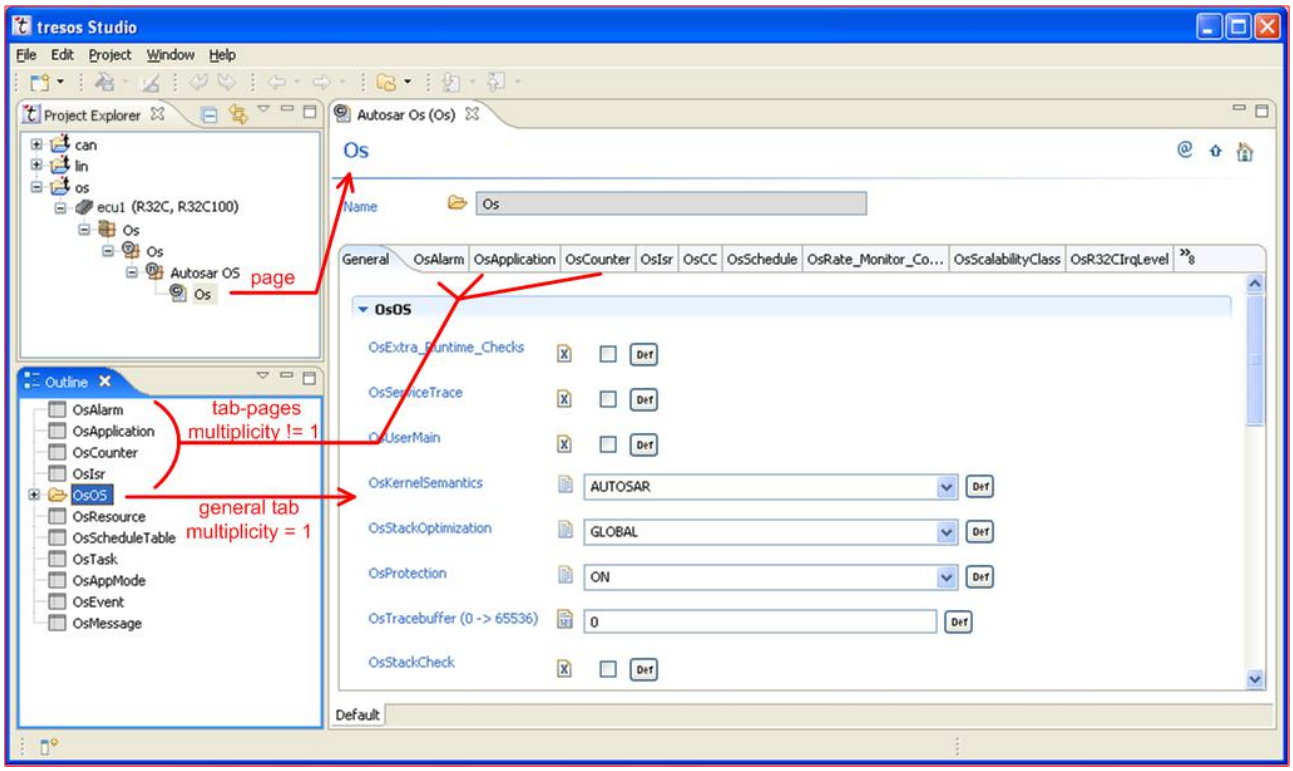


Figure 5.24. MODULE-DEF in the editor

Double clicking the index column in a table that contains containers switches the editor to a new page representing the container.

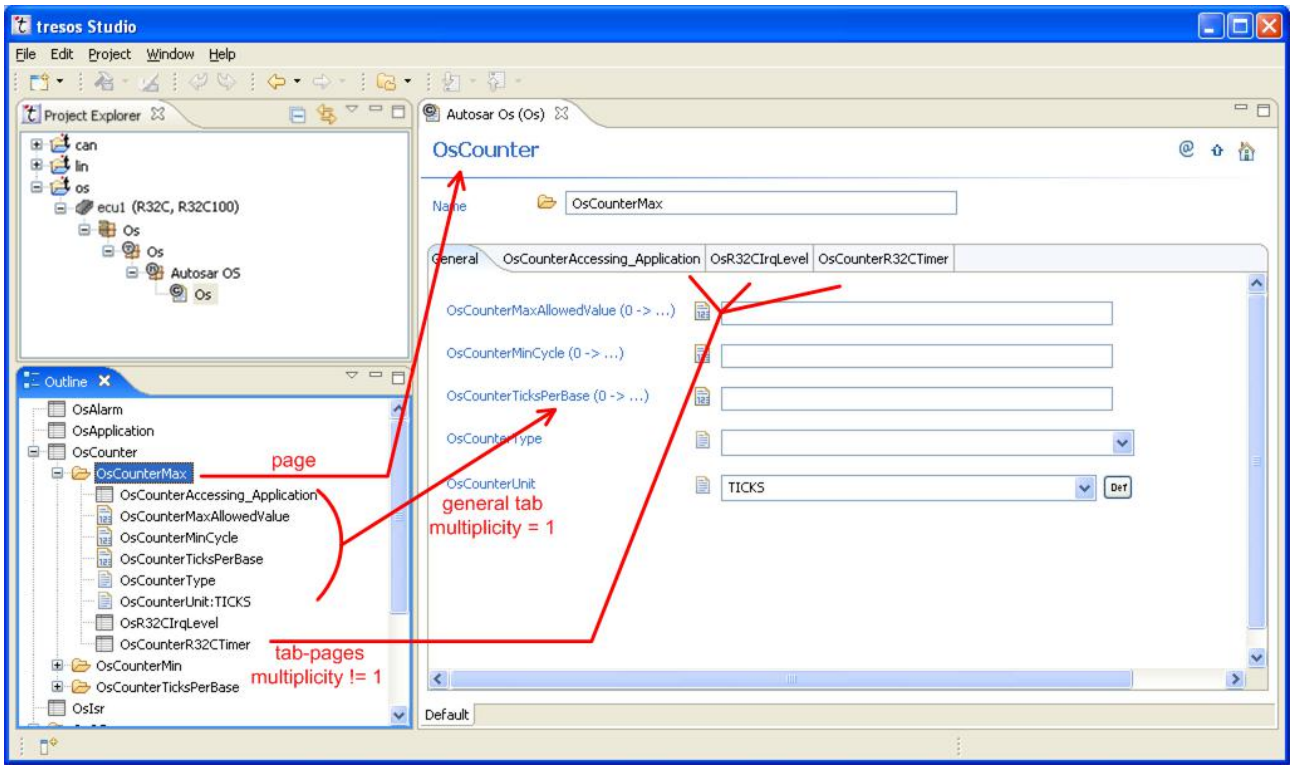


Figure 5.25. PARAM-CONF-CONTAINER-DEF in the GUI

5.11.2. Module GUI configuration

There are some parameters which change the GUI-representation of a whole module. Thus, these parameters are stored in the file "plugin.xml" of the module.

```
<extension point="dreisoft.tresos.launcher2.plugin.configuration"
           id="ConfigId" name="Configuration">
  <editor id="EditorId">
    <class
        class="dreisoft.tresos.launcher2.editor.GenericConfigEditor">
      <parameter name="schema" value="ASPath:/TS_T17D9M2I0R0/EcuC"/>
      <parameter name="noLinks" value="false"/>
      <parameter name="groupLinks" value="false"/>
      <parameter name="groupContainers" value="true"/>
      <parameter name="groupTables" value="true"/>
      <parameter name="optionalGeneralTab" value="true"/>
```

```
    </class>
  </editor>
</configuration>
```

Parameter	Default	Description
noTabs	false	If set to true tables will not be placed on own tabs.
noLinks	false	Do not create forward and backward hyperlinks. For each element which is not shown on the tab of its enclosing node (e.g. when the TAB-attribute is used or for tables), a hyperlink is shown between its sibling elements pointing to the tab where the element is shown. Another hyperlink is created on the tab where the element is shown which is pointing back.
groupLinks	false	Group all hyperlink elements after all configuration parameters instead of creating the hyperlinks at the position where the element occurs in the schema (between its sibling elements).
groupContainers	true	Group all containers after variables and references. Usually, each element is shown at the position where the element occurs in the schema (between its sibling elements). Setting this option to true renders containers after variables, references and tables.
groupTables	true	Group all tables after the variables and references. Usually, each element is shown at the position where the element occurs in the schema (between its sibling elements). Setting this option to true renders tables after variables and references.
optionalGeneralTab	true	Omit General Tab if only links would be placed on it.

Table 5.40. Module GUI attributes

5.11.3. GUI Elements

The following chapter describes the various elements in a parameter definition and how they are shown in the GUI.

The visualization of the elements in the GUI can be tailored by setting respective attributes in the XDM file. The chapter will outline all these attributes.

5.11.3.1. Attributes

There are several attributes which can be added to a schema-file to enrich the GUI.

	TOOLTIP	VISIBLE	DESC	EDITABLE	LABEL	FRAME	LAYOUT	TITLE	COLUMNS	COLUMN_TITLE	COLUMN_TOOLTIP	WIDTH	TAB	DEFAULT_RADIX
Boolean values	x	x	x	x	x								x	
Integer values	x	x	x	x	x							x	x	x
Float values	x	x	x	x	x							x	x	
String values	x	x	x	x	x							x	x	
Enumerations	x	x	x	x	x							x	x	
References	x	x	x	x	x							x	x	
Container		x	x		x	x	x	x					x	
Choices	x	x	x	x	x	x	x					x	x	
Lists	x	x	x	x	x				x	x	x		x	

Figure 5.26. Attributes

Note

Note that an automatic attribute can be used instead of a fixed value for all data-attributes (tag <a:da/>).

```
<a:da name="<attribute_name>" type="XPath" expr="../Integer2 &lt; 20"/>
```

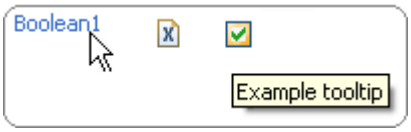
For schematic-attributes (tag <a:a/>) this is not possible, so for instance LABEL you can't use automatic attributes.

5.11.3.2. Boolean values

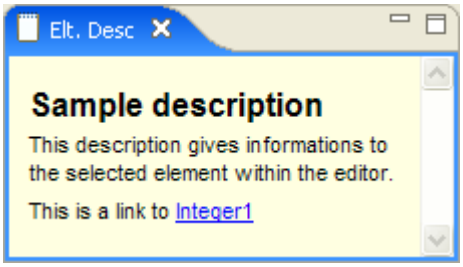
Each boolean value defined within the schematic-xdm will be represented by a checkbox in the editor.

```
<v:var name="Boolean1" type="BOOLEAN"/>
```

Figure 5.27. Checkbox

Attribute	Description
TOOLTIP	Used to add a tooltip to the checkbox. `<a:da name="TOOLTIP" value="Example tooltip"/>`  Figure 5.28. Checkbox Tooltip
VISIBLE	With this attribute, you can hide the checkbox within the editor. `<a:a name="VISIBLE" value="false"/>`

Attribute	Description
DESC	

Attribute	Description
	Used to add a description for the boolean value. The description is visible for the user within the "Elt. Desc"-view, if the checkbox is selected. As you can see in the following examples, there can be single-line, multi-line or html-description optional containing links to other configuration elements. A click on the link will show the target-element within the editor. `<a:a name="DESC" value="The parameter depicts..."/>` `<a:a name="DESC"><a:v>` `  Example description` `  taking multiple lines` `</a:v></a:a>` `<a:a name="DESC"><a:v><![CDATA[` `<html>` `  <h1>Sample description</h1>` `  This description gives informations about` `  the selected element within the editor. ` `  <p>` `  This is a link to <a href="path:/AUTOSAR/TOP-LEVEL-` `  PACKAGES/Can/ELEMENTS/ModulePlugIn/Container/Integer1"/>` `  Integer1</a>` `</html>` `] ]></a:v></a:a>`  Figure 5.29. Checkbox Description To create a link, you have to prefix the XPath of the target-element with "path:" and use this as the value of the "href"-tag as shown in the example. **Note** To get the path to a configuration element, just select it. Then, its XPath

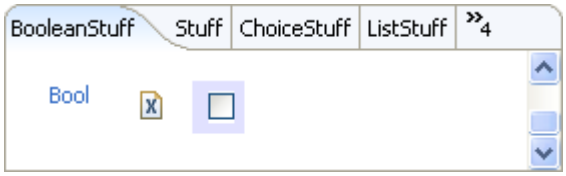
Attribute	Description
	is shown in the "Elt. Info" Element Outline View.
EDITABLE	Used to allow the user to edit the element or not. If this attributes value is set to "false", the checkbox will be greyed out. `<a:da name="EDITABLE" value="false"/>` Figure 5.30. Non-editable Checkbox
LABEL	Used to change the label of the checkbox. `<a:a name="LABEL" value="Example label"/>` Figure 5.31. Checkbox Label
TAB	Via this attribute, an element can be placed on a different tab than its default. `<a:a name="TAB" value="BooleanStuff"/>`  Figure 5.32. Checkbox TAB

Table 5.41. Boolean Attributes

5.11.3.3. Integer Values, Float Values, String Values, Enumerations and References

Enumerations and references are represented by a combobox. Integer values, float values, and string values are represented by a text field within the editor.

```
<v:var name="Integer1" type="INTEGER">
<v:var name="Float1" type="FLOAT">
<v:var name="String1" type="STRING">
<v:ref name="Reference1" type="REFERENCE">
<v:var name="Enumeration1" type="ENUMERATION">
```

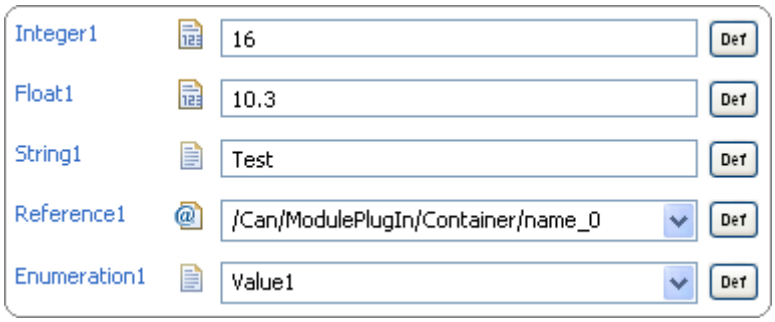
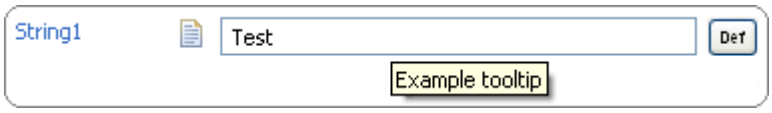
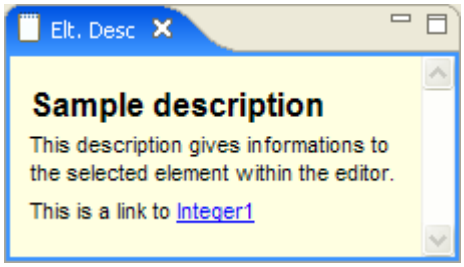


Figure 5.33. Field elements

Attribute	Description
TOOLTIP	Used to add a tooltip to the field. `<a:da name="TOOLTIP" value="Example tooltip"/>`  Figure 5.34. Tooltip for Parameters
VISIBLE	With this attribute, you can hide the field within the editor. `<a:a name="VISIBLE" value="false"/>`
DESC	Used to add a description for the configuration element. The description is visible for the user within the "Elt. Desc" Element Outline View, if the checkbox is selected. See Section 5.11.3.2: Boolean values for details, how to create descriptions. As you can see there, you can create single-line, multi-line or html-descriptions containing links to other configuration elements.  Figure 5.35. Parameter Description To create a link, you have to prefix the XPath of the target-element with "path:" and use this as the value of the "href"-tag as shown in the example. **Note** To get the path to an configuration element, just select it. Then, its XPath is shown in the "Elt. Info" Element Outline View.

Attribute	Description
EDITABLE	Used to allow the user to edit the element or not. If this attribute value is set to "false", the checkbox will be greyed out. `<a:da name="EDITABLE" value="false"/>` Figure 5.36. Non-editable Parameter
LABEL	Used to change the label of the field. `<a:a name="LABEL" value="Example label"/>` Figure 5.37. Parameter Label
WIDTH	The width of a text field or a combobox in terms of characters. `<a:a name="WIDTH" value="10"/>` Figure 5.38. Parameter Width

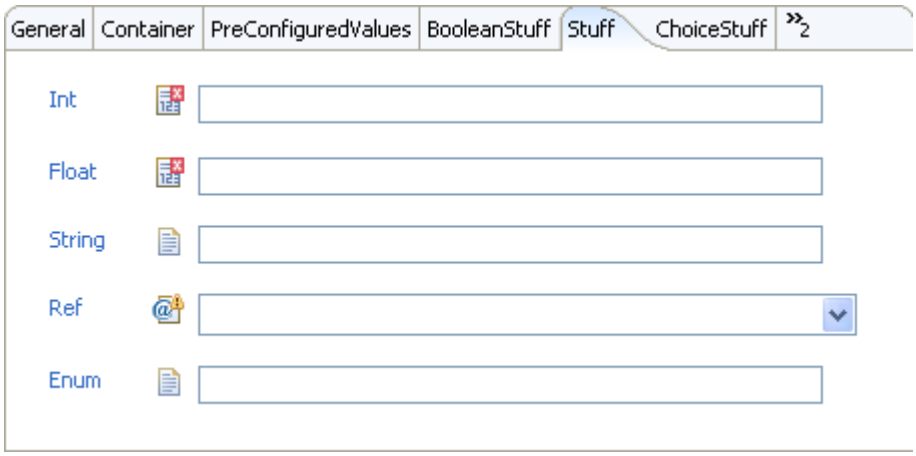
Attribute	Description
TAB	Via this attribute, an element can be placed on a different tab than its default. `<a:a name="TAB" value="Stuff"/>`  Figure 5.39. Int Float TAB
DEFAULT_RADIX	Sets an initial numeric representation for the GUI element of an `INTEGER` node. Valid values for the `DEFAULT_RADIX` attribute are: `DEC` (decimal, default), `HEX` (hexadecimal), `OCT` (octal) and `BIN` (binary). `<a:a name="DEFAULT_RADIX" value="HEX" />` Figure 5.40. DEFAULT_RADIX attribute for INTEGER nodes

Table 5.42. Parameter and Reference Attribute

5.11.3.4. Container

A container creates a new section within the editor. A container should always contain other configuration elements.

```
<v:ctr name="Container" type="IDENTIFIABLE">
```

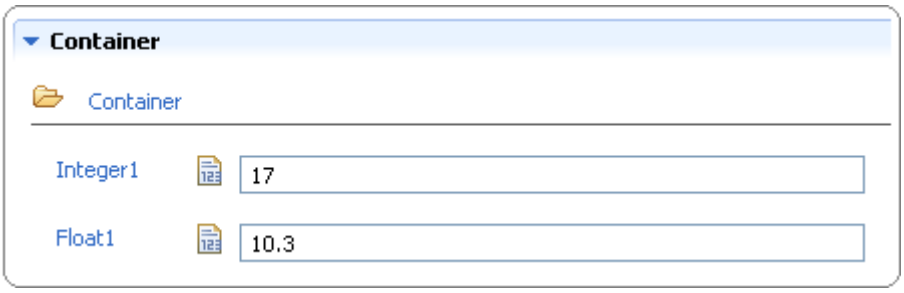
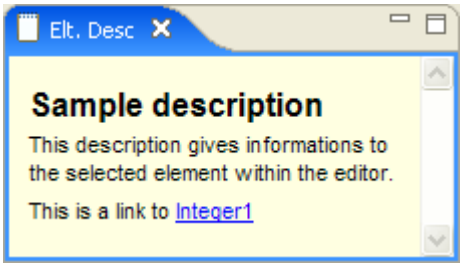
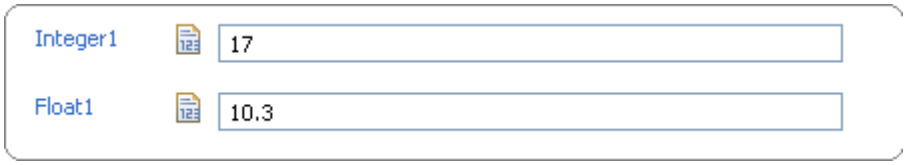
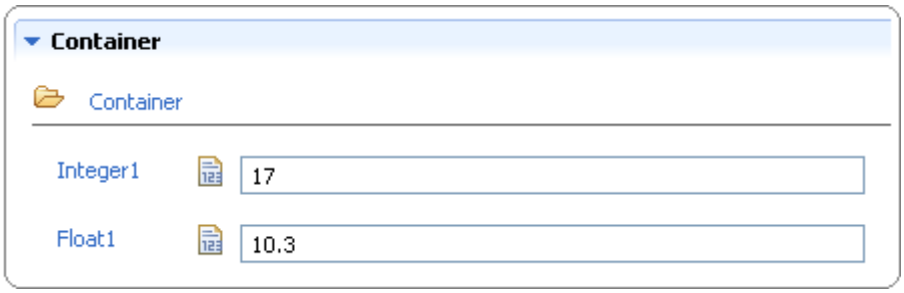
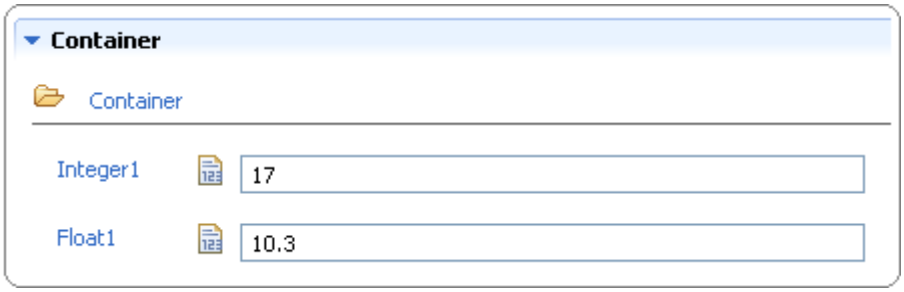
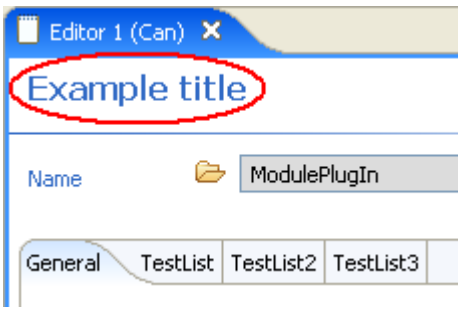


Figure 5.41. Container configuration element

Attribute	Description
VISIBLE	With this attribute, you can hide the container within the editor. `<a:a name="VISIBLE" value="false"/>`
DESC	Used to add a description for the configuration element. The description is visible for the user within the "Elt. Desc" Element Outline View, if the checkbox is selected. See Section 5.11.3.2: Boolean values for details, how to create descriptions. As you can see there, you can create single-line, multi-line or html-descriptions containing links to other configuration elements.  Figure 5.42. Container Description To create a link, you have to prefix the XPath of the target-element with "path:" and use this as the value of the "href"-tag as shown in the example. **Note** To get the path to an configuration element, just select it. Then, its XPath is shown in the "Elt. Info" Element Outline View.
LABEL	Used to change the label of the container. `<a:a name="LABEL" value="Example label"/>` Figure 5.43. Container Label

Attribute	Description
FRAME	Used to change the frame around the configuration element. Possible values are: ▶ `NONE` no frame. ▶ `LINE` lined border with a title. ▶ `TITLE`(default) lined border with a title which can be expanded and collapsed by a click on the title. `<a:da name="FRAME" value="NONE"/>`  Figure 5.44. FRAME with value NONE Figure 5.45. FRAME with value LINE  Figure 5.46. FRAME with value TITLE

Attribute	Description
LAYOUT	Used to change the layout of the configuration element. Possible values are: ▶ `HORIZONTAL` layouts the elements inside the container horizontal. ▶ `VERTICAL`(default) layouts the elements inside the container vertical. `<a:a name="LAYOUT" value="HORIZONTAL"/>` Figure 5.47. LAYOUT with value HORIZONTAL  Figure 5.48. LAYOUT with value VERTICAL
TITLE	Used to set the title of a page. You can only change the title of containers which creates a new page. `<a:da name="TITLE" value="Example title"/>`  Figure 5.49. Title of a Container

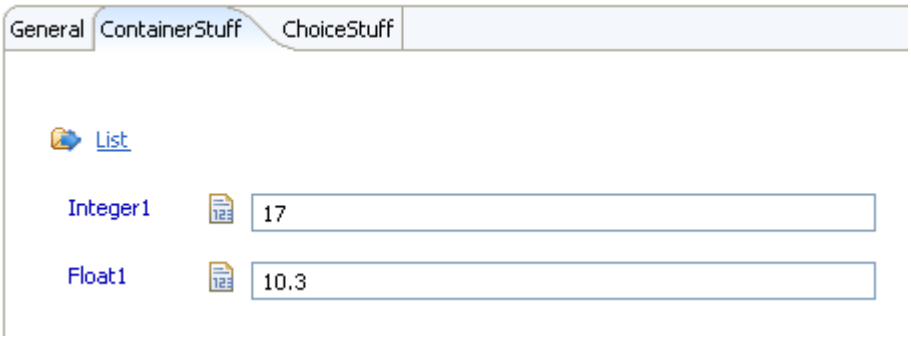
Attribute	Description
TAB	Via this attribute, an element can be placed on a different tab than its default. `<a:a name="TAB" value="ContainerStuff"/>`  Figure 5.50. Container TAB

Table 5.43. Container Attributes

5.11.3.5. Choices

A choice creates a new section within the editor which can be expanded and collapsed by a click on the title. The first element within a choice is a combobox followed by the choosable configuration elements itself. Changing the selection of the combobox will enable the selected configuration element below. The others stay disabled.

```
<v:chc name="Choice1">
```

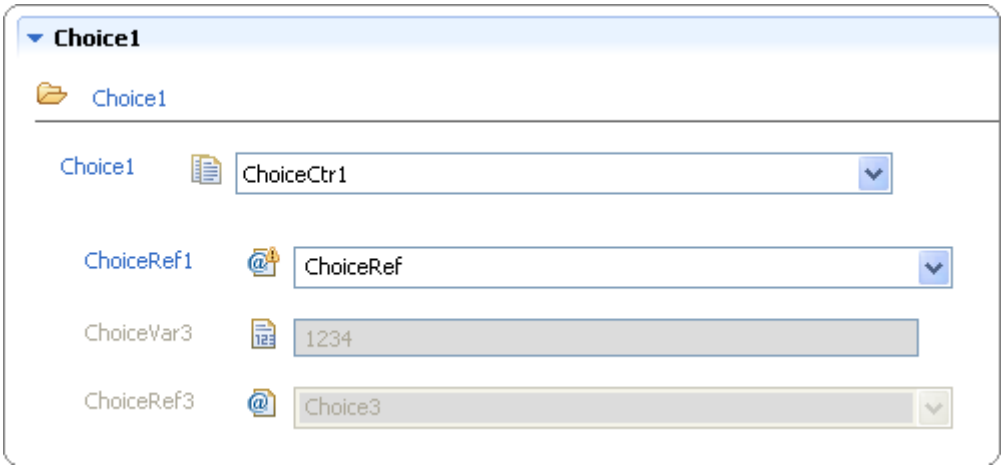
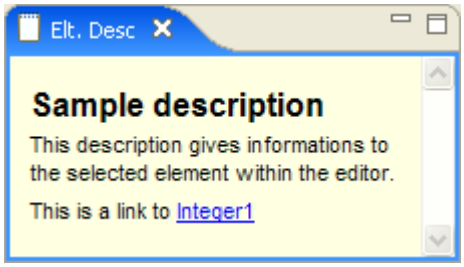
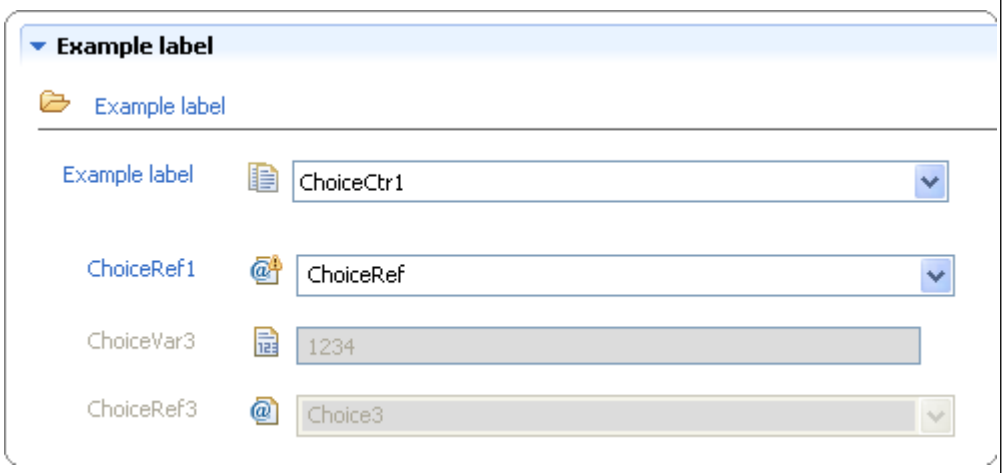
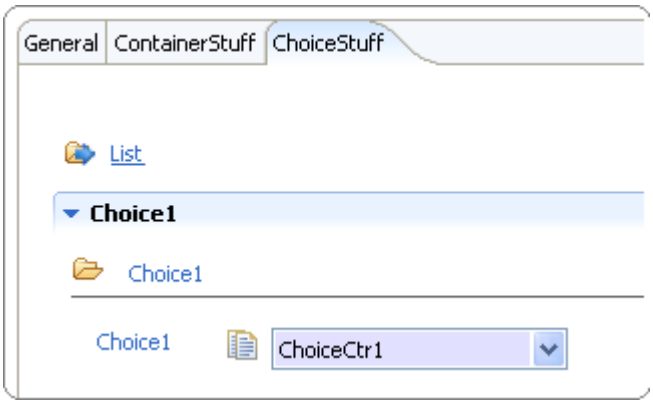


Figure 5.51. Choice configuration element

Attribute	Description
TOOLTIP	Used to add a tooltip to the choice. `<a:da name="TOOLTIP" value="Example Tooltip"/>` Figure 5.52. Choice Tooltip
VISIBLE	With this attribute, you can hide the choice within the editor. `<a:a name="VISIBLE" value="false"/>`
DESC	Used to add a description for the configuration element. The description is visible for the user within the "Elt. Desc" Element Outline View, if the checkbox is selected. See Section 5.11.3.2: Boolean values for details, how to create descriptions. As you can see there, you can create single-line, multi-line or html-descriptions containing links to other configuration elements.  Figure 5.53. Description of a Choice To create a link, you have to prefix the XPath of the target-element with "path:" and use this as the value of the "href"-tag as shown in the example. **Note** To get the path to an configuration element, just select it. Then, its XPath is shown in the "Elt. Info" Element Outline View.

Attribute	Description
EDITABLE	Used to allow the user to edit the element or not. If this attributes value is set to "false", the checkbox will be greyed out. `<a:da name="EDITABLE" value="false"/>` Figure 5.54. Non-editable Choice
LABEL	Used to change the label of the field. `<a:a name="LABEL" value="Example label"/>`  Figure 5.55. Choice Label
WIDTH	The width of the choice in terms of characters. `<a:a name="WIDTH" value="20"/>` Figure 5.56. Choice Width

Attribute	Description
TAB	Via this attribute, an element can be placed on a different tab than its default. `<a:a name="TAB" value="ChoiceStuff"/>`  Figure 5.57. Choice TAB

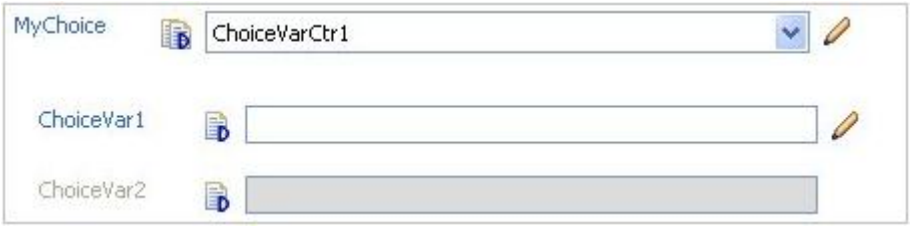
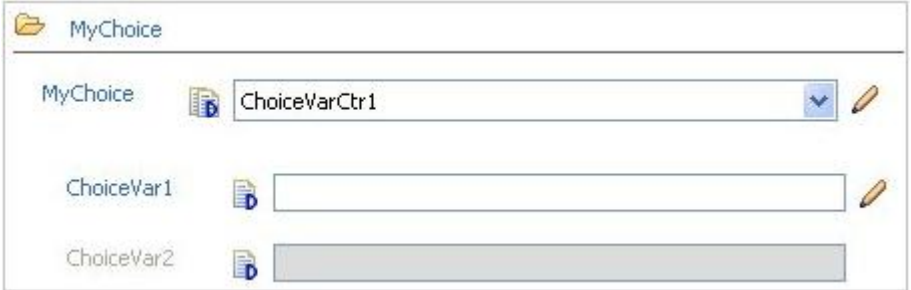
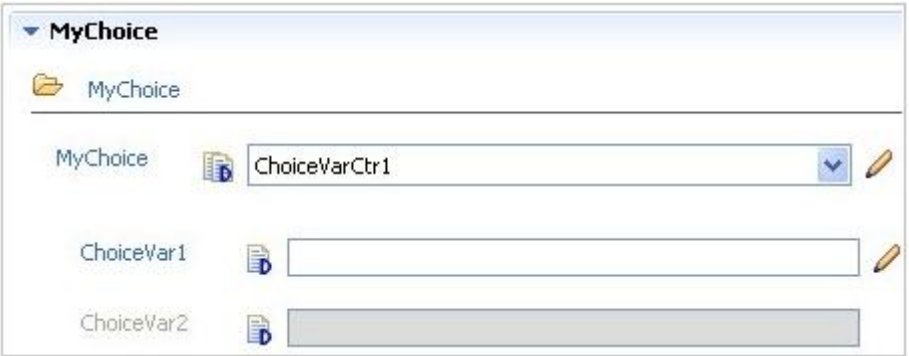
Attribute	Description
FRAME	Used to change the frame around the configuration element. Possible values are: ▶ `NONE` no frame. ▶ `LINE` lined border with a title. ▶ `TITLE`(default) lined border with a title which can be expanded and collapsed by a click on the title. `<a:da name="FRAME" value="NONE"/>`  Figure 5.58. FRAME with value NONE  Figure 5.59. FRAME with value LINE  Figure 5.60. FRAME with value TITLE

Table 5.44. Choice attributes

5.11.3.6. Lists

Each list gets its own tab-page within the editor and each configured value gets its own row within the list. Per default, the index (position within the list) and the name of the configured element gets an own column within the list.

```
<v:lst name="TestList" type="MAP">
  <v:ctr name="ListContainer1" type="IDENTIFIABLE">
    <v:var name="ListString" type="STRING"/>
    <v:var name="ListInteger" type="INTEGER"/>
  </v:ctr>
</v:lst>
```

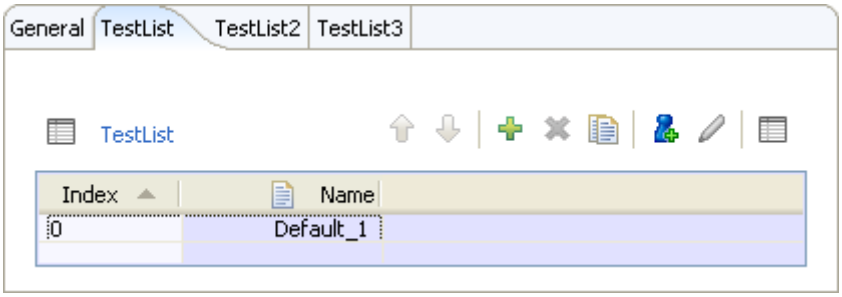
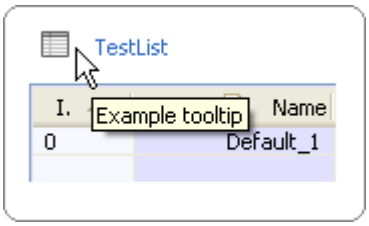
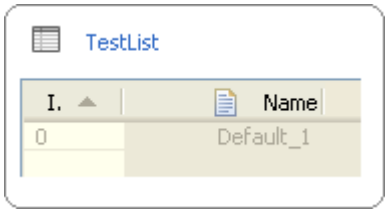
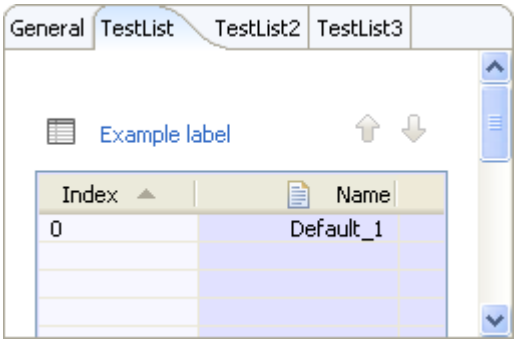
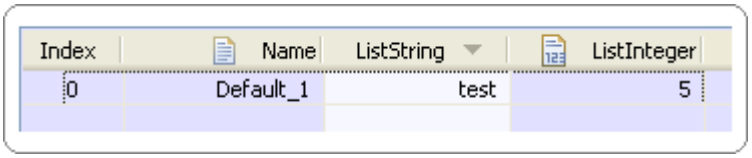


Figure 5.61. List configuration element

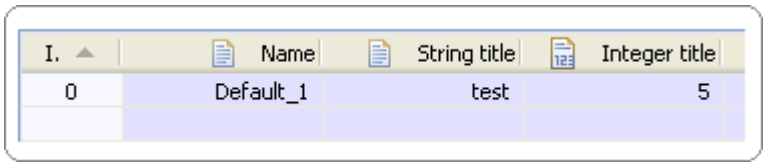
Attribute	Description
TOOLTIP	Used to add a tooltip to the configuration element. `<a:da name="TOOLTIP" value="Example tooltip"/>`  Figure 5.62. Tooltip inside a list
VISIBLE	With this attribute, you can hide the tab-page of the list within the editor. `<a:a name="VISIBLE" value="false"/>`
DESC	Used to add a description for the configuration element. The description is visible for the user within the "Elt. Desc" Element Outline View, if the list icon is clicked. See Section 5.11.3.2: Boolean values for details, how to create descriptions. As you can see there, you can create single-line, multi-line or html-descriptions containing links to other configuration elements. Figure 5.63. Description of a List

Attribute	Description
EDITABLE	Used to disable the configuration list. **Note** You cannot use the editor to select a list-entry of a disabled list. `<a:da name="EDITABLE" value="false"/>`  Figure 5.64. Editable list
LABEL	Used to change the label of the list element. `<a:a name="LABEL" value="Example label"/>`  Figure 5.65. Label of a list

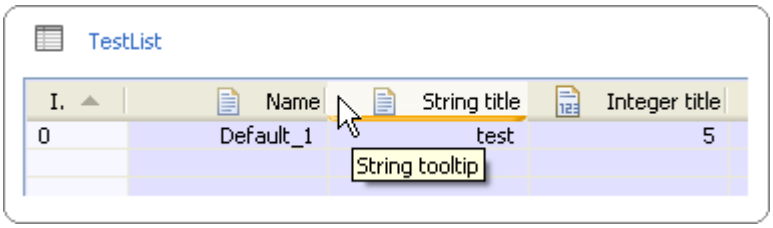
Attribute	Description

COLUMNS	Columns of a table are defined via the `COLUMNS` attribute if the list contains containers or choices. The values of the attribute can reference parameters or references (that have `UPPER-MULTIPLICITY` = `LOWER_MULTIPLICITY` == 1) relative to the container or choice inside the list via an XPath.
	
	`<a:a name="COLUMNS">`
	`  <a:v>ListString</a:v>`
	`  <a:v>SubContainer/ListInteger</a:v>`
	`</a:a>`
	
	 
	
	Figure 5.66. List Columns

Attribute	Description

COLUMN_TITLE	Used to set the title of a custom column within a table.
	
	`<v:lst name="TestList" type="MAP">`
	`  <a:a name="COLUMNS">`
	`  <a:v>ListString</a:v>`
	`  <a:v>ListInteger</a:v>`
	`  </a:a>`
	
	`  <v:ctr name="ListContainer1" type="IDENTIFIABLE">`
	`  <v:var name="ListString" type="STRING">`
	`  <a:a name="COLUMN_TITLE" value="String title"/>`
	`  <a:a name="COLUMN_TOOLTIP" value="String tooltip"/>`
	`  </v:var>`
	`  <v:var name="ListInteger" type="INTEGER">`
	`  <a:a name="COLUMN_TITLE" value="Integer title"/>`
	`  <a:a name="COLUMN_TOOLTIP" value="Integer tooltip"/>`
	`  </v:var>`
	`  </v:ctr>`
	`</v:lst>`
	
	 
	
	Figure 5.67. Column Title

Attribute	Description

COLUMN_TOOLTIP	Used to add a tooltip for a custom column within a table.
	
	`<v:lst name="TestList" type="MAP">`
	`  <a:a name="COLUMNS">`
	`  <a:v>ListString</a:v>`
	`  <a:v>ListInteger</a:v>`
	`  </a:a>`
	
	`  <v:ctr name="ListContainer1" type="IDENTIFIABLE">`
	`  <v:var name="ListString" type="STRING">`
	`  <a:a name="COLUMN_TITLE" value="String title"/>`
	`  <a:a name="COLUMN_TOOLTIP" value="String tooltip"/>`
	`  </v:var>`
	`  <v:var name="ListInteger" type="INTEGER">`
	`  <a:a name="COLUMN_TITLE" value="Integer title"/>`
	`  <a:a name="COLUMN_TOOLTIP" value="Integer tooltip"/>`
	`  </v:var>`
	`  </v:ctr>`
	`</v:lst>`
	
	 
	
	Figure 5.68. Column Tooltip

TAB	Via this attribute, an element can be placed on a different tab than its default. `  <a:a name="TAB" value="ListStuff"/>`

Table 5.45. List Attributes

5.12. The Code Generator

The DataModel provides a sophisticated *Code Generator* which takes a code template, enhanced with special macros which refer to variables in the Configuration Data, and which generates code.

The following chapter explains the macro language with which code can be generated dynamically.

5.12.1. General concepts

The code generator takes an input stream, performs macro expansion and writes the result in an output stream. All macros used by the code generator are parameterized by the configuration data. The macros can access the data with an XPath expression which allows them to do mathematical evaluations, list and string manipulation, boolean evaluations and list operations.

The code template read by the code generator is a simple ASCII-file. Macros can be introduced anywhere in the template. Macro-expressions are enclosed by **[!** and **!]** (if not stated otherwise). For example:

```
[!INCLUDE "test.m"!]
```

includes the file `test.m`. All macro-expressions from **[!** till **!]** are eliminated from the generated code.

All code that should be generated must be between the two tags:

```
[!CODE!]
...
[!ENDCODE!]
```

All text before **[!CODE!]** and all text after **[!ENDCODE!]** will not be written in the output stream.

The need for the **[!CODE!]** can be disabled via the Generator-API.

In addition to the **[!CODE!]**-tag the Generator also supports the tags **[!NOCODE!]** and **[!ENDNOCODE!]** which can be used inside the **[!CODE!]**-tag to temporarily disable code printout. Only variables and macros are evaluated inside these tags.

The code generator supports a special line quoting which allows to join lines before macros are evaluated. A line end can be quoted with **[!//** at the end of the line.

5.12.1.1. Addressing nodes using XPath

Several generator-commands take xpath-expressions which can be used to address configuration-nodes. For this, the correct XPath pointing to the node must be entered. As described in chapter Section 5.9.3.1: Path in Template, the EB tresos Studio configurator provides a view showing the complete XPath to the selected configuration-node within the current editor.

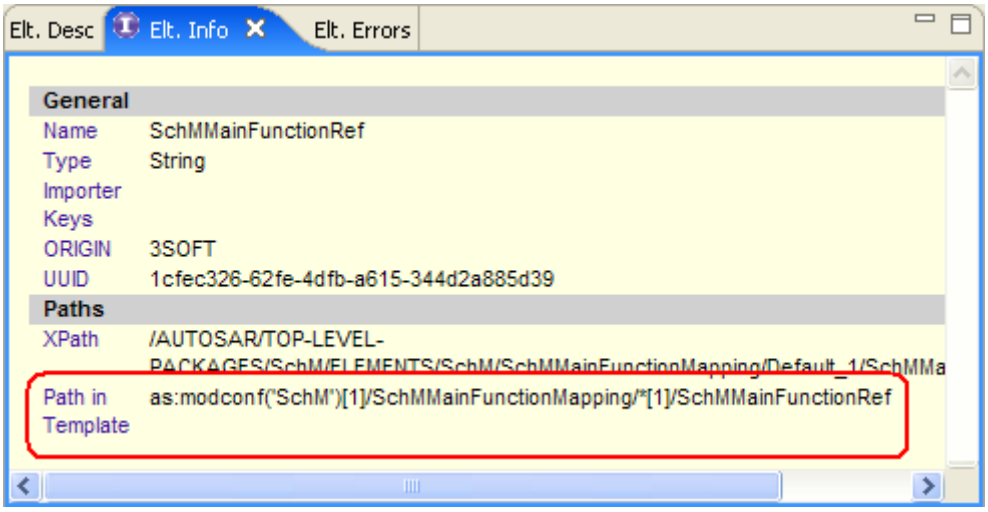


Figure 5.69. Path in template

5.12.1.2. Identical generated files

The generator will never overwrite an existing file, if the contents do not change. For this, a comparison on a per-line basis is done before writing to the target-file. To disable the comparison for single lines only, the line must contain the following keyword (with exclamation marks and capital letters):

```
!!!IGNORE-LINE!!!
```

There may be any other text before and after this keyword.

5.12.2. Codetemplate Console

To use the **Codetemplate Console**, you have to enable the preference "Enable developer features" using "Windows/Preferences.../Tresos Studio".

You can open the **Codetemplate Console** from the context-menu of a node within the **Outline view** by choosing "Codetemplate console". The selected node will be used as context-node for all executed templates.

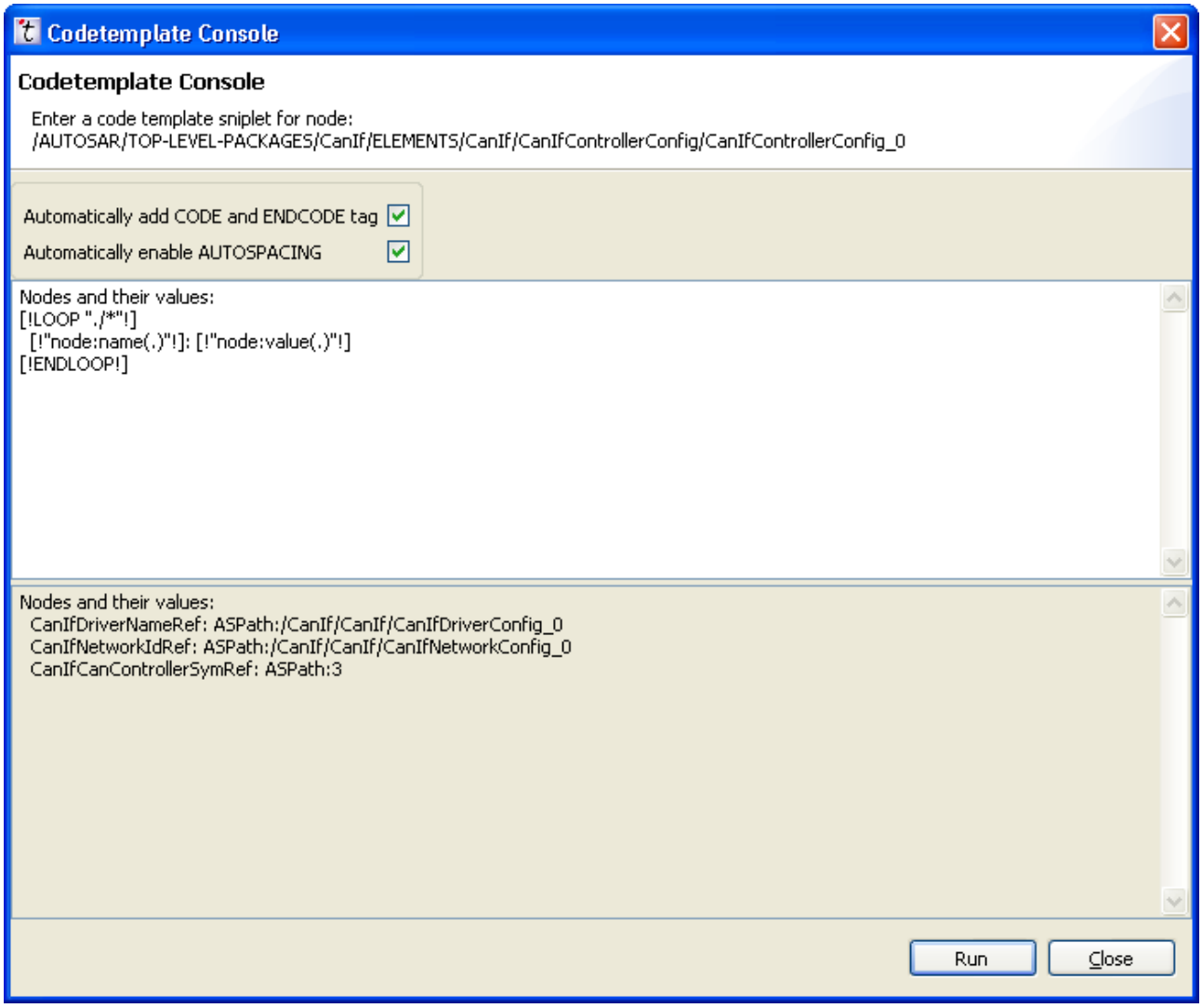


Figure 5.70. Codetemplate Console

Type or paste any code template into the upper text area and click "Run" to evaluate it. The result will be shown in the lower text area.

5.12.3. Comments

The simplest macro expression is the comment. There are two forms of comments:

- Line Comments
- Multi-line comments

Line comments start with **[!//** and expand to the end of the line. All text starting with **[!//** is not written in the output file.

If a line ends with the sequence **[!//** the line termination (linefeed/ctrl-linefeed) is quoted.

A multi-line comment starts with **[!/*** and ends with ***/!]**. A Multi-line comment can span multiple lines. All text from **[!/*** to ***/!]** is removed from the output file.

```
[!/*
multiline-comment
*/!]
... [!// line-comment
```

5.12.4. Evaluating variables and expressions

Variables from the configuration data and XPath expressions can be evaluated with:

```
[!"<Expression>"!]
```

The Expression is an XPath expression evaluated on the configuration so that all variables in the configuration data tree can be accessed.

5.12.5. Including other files

Other files can be included with the include statement:

```
[!INCLUDE "<Filename>"!]
```

The include expression is substituted by the contents of the included file. Macro expansion is also performed on the included file.

If the filename starts with a question mark, the rest of the filename is interpreted as an XPath expression which results in the filename to be loaded.

If the include expression is inside an if-expression the file is only included if the if-expression evaluates to true. Likewise an include expression in a loop-expression includes the file each time the loop runs.

5.12.6. Conditions

With conditions parts of the code can be excluded from the output stream. There are two types of conditions:

- if-constructs
- end-conditions

If-constructs are conditional expressions as known in most programming languages. The syntax is as follows:

```
[!IF "<Expression>"!]
...
[!ELSEIF "<Expression>"!]
...
[!ELSE!]
...
[!ENDIF!]
```

The **[!ELSEIF!]**-clause and the **[!ELSE!]**-clause are optional. There can be more than one **[!ELSEIF!]**-clause. The expression used in the **[!IF!]**- and the **[!ELSEIF!]**-clause is an XPath-expression which must result in a boolean value.

The second form of a condition, the end-condition, is a short form that is used to exclude a single line from the output stream. The end- condition is always at the end of the line:

```
... [!WHEN "<Expression>"!]
```

If the expression which is an XPath-expression evaluates to 'false', the complete line is omitted from the output stream.

5.12.7. Loops and selects

The XPath language allows to select a set of nodes, not only one single value.

The code generator supports to iterate over such node-sets. While iterating, the current node pointer is set once to each node of the selected node-set. The syntax of such an iteration is:

```
[!LOOP "<Expression>"!]
...
```

```
[!ENDLOOP!]
```

For each node in the node-set, the code block (eventually instrumented with macros) between **[!LOOP!]** and **[!ENDLOOP!]** is written to the output stream. For each run through the loop, the current node pointer is set to a new node in the node set. Thus inside the loop relative XPath expressions (in relation to the selected node) can be used.

```
[!SELECT "<Expression>"!]
...
[!ENDSELECT!]
```

The select-expression can be used to execute contained XPath-expressions relative to a single (the currently selected) node. Hint: The select-expression behaves identical to the loop.

In addition to looping over a group of nodes the code generator also allows looping over numbers with the for-statement of other languages. The code generator for statement has the following syntax:

```
[!FOR "<Variable-Name>" = "<From-Value>"
TO "<To-Value>"!]
...
[!ENDFOR!]
```

```
[!FOR "<Variable-Name>" = "<From-Value>"
TO "<To-Value>" STEP "<Step-Value>"!]
...
[!ENDFOR!]
```

The for-statement steps the values given by the XPath expression **<From-Value>** to the value of the XPath expression given by the **<To-Value>** setting the variable **<Variable-Name>** in each iteration. The value of this variable increases (or decreases) by the **<Step-Value>** in each iteration. The **<Step-Value>** defaults to 1.

The variable **<Variable-Name>** is a local variable bound to the scope of the for-statement.

5.12.8. Leaving loops

The following loops can be left by using the BREAK-statement:

- FOR

- LOOP
- SELECT

Example:

```
[!FOR "x" = "0" TO "10"!]
  [!IF "$x>3"!][!BREAK!][!ENDIF!]
  [!"."!]
[!ENDFOR!]
```

In the example above, the FOR-loop will be left, if the variable "x" is greater than 3.

5.12.9. Variables and macros

The code generator allows to define variables inside the code template which can be used in XPath expressions.

Variables are defined in the following way:

```
  [!VAR "<Variable-Name>" = "<Expression>"!]
  [!VAR "<Variable-Name>"!]
  <Expression>
  [!ENDVAR!]
```

The first variable-definition maps a variable name to an XPath expression. The XPath expression is evaluated at the point where the variable is defined.

In the second variable-definition, the variable value can be set in a much more complex way. The expression in this form can contain other macro expressions. Also in this form the expression is evaluated once at the point where the variable is defined.

Variables defined in this form can be used in any XPath expression as a variable in the form **$<Variable-Name>**.

Variables are always defined globally which means they are defined from their first occurrence until the generator finishes the code generation.

Besides the user defined variables in the code template, there exists some variables which are always set and can be used without defining them.

datetime
: Generator variable storing the current timestamp in format YYYY-MM-DD hh:mm:SS.

date
: Generator variable storing the current date in format YYYY-MM-DD.

time

Generator variable storing the current time in format HH:MM:SS.

target

Generator variable storing the target, set by the preferences or the commandline. In legacy mode the target can be set with `-Dtarget=<target>`.

derivate

Generator variable storing the derivate, set by the preferences or the commandline. In legacy mode the derivate can be set with `-Dderivate=<derivate>`.

ecuid

Generator variable storing the ecuid, set by the preferences or the command line. In legacy mode the ecuid can be set with `-Decuid=<ECU-Id>`.

variant

Generator variable that stores the variant name of EB tresos Studio. E.g. `2007.b`, `2008.a`...

project

Generator variable that stores the name of the project currently processed. In legacy mode this variable can be set with `-Dproject=<Project Name>`.

module

Generator variable that stores the module-id of the module currently processed.

modules

Generator variable that stores a space separated list of all module-id the a currently processed in this generator or verify run.

generateModules

Generator variable that stores a space separated list of all module-ids for which code gets generated in this generator run.

enabledModules

Generator variable that stores a space separated list of all module-ids which are enabled in the processed project.

defaultconfig

Generator variable that stores the name of the module configuration that should be considered as the default configuration when there are more than one module configuration for one module generated. This is by default the first found module configuration.

configs

Generator variable that stores the name of all module configurations of the module currently processed.

generateConfigs

Generator variable that stores the name of all module configurations of the module currently processed that are enabled.

moduleSpecVer

Generator variable that stores the specification version of the module currently processed in the form `<major>.<minor>.<patch>`.

moduleSwVer

Generator variable that stores the software version of the module currently processed in the form `<major>.<minor>.<patch>`.

moduleType

Generator variable that stores the module type of the module currently processed.

moduleLayer

Generator variable that stores the module layer of the module currently processed.

moduleCategory

Generator variable that stores the module category of the module currently processed.

mode

Generator variable that stores the generator mode.

Besides variables the code generator supports macros. Macros are defined with:

```
[!MACRO "<Macro-Name>",
        "<Parameter-Name>" = "<Default-Value>",...!]
[!ENDMACRO!]
```

Macros can have parameters which can be used like variables in XPath expressions inside the macro. All parameters have a name and can have a default value which is used when the parameter is not set in the macro-call. The default-value is always interpreted as an XPath expression. The parameter-name is a normal text string. The macro name is also a normal text string if it does not start with a question mark, in which case the reset of the macro name is interpreted as an XPath expression.

A macro can be called with the following command:

```
[!CALL "<Macro-Name>",
       "<Parameter-Name>" = "<Parameter-Value>",...!]
```

The content of the macro is evaluated when it is called and added to the output stream at the position of the call.

Each parameter value is interpreted as an XPath expression. Each parameter-name is a plain text string. Like in the macro definition the macro-name normally is a plain text string if it does not start with a question mark. In this case it is also interpreted as an XPath expression.

All parameters of a macro are interpreted as local variables which means they shadow global variables during the macro call and disappear after the macro finishes.

5.12.10. Spaces, newlines and indentation

5.12.10.1. Tab stops

Indentation can be controlled with the special variable **$tabs**. This variable can be used as either a parameter to a macro or as a normal variable. When the Code Generator encounters a percent sign at the beginning of a line, followed by three spaces, this character combination is substituted by a number of spaces equal to **$tabs**.

```
%   intended text
[!"$tabs"!]intended text
```

5.12.10.2. Whitespaces

To explicitly write whitespaces, you can use the WS-command which optionally gets the number of spaces as parameter:

```
[!WS!]one whitespace
[!WS "2"!]two spaces
[!WS "$tabs"!]spaces as defined in variable tabs
```

5.12.10.3. Newlines

To explicitly write newlines, you can use the CR-command (carriage return):

```
[!CR!]
```

5.12.10.4. Auto-spacing

Autospacing allows you to indent your generator-commands within your code-templates without influencing your generated code with the spaces of the indentation and the newlines.

Autospacing can be switched on for each template-file by a single command:

```
[!AUTOSPACING!]
```

If switched on it does the following for commands which do not produce output:

- if there are only spaces in front of the command, they will be removed
- if there are only spaces and a newline after the command, the spaces and the newline will be removed

Suppose the following code-template once with and once without autospacing:

```
[!CODE!]
  [!LOOP "text:split('1 2')"!]
    myoutput
  [!ENDLOOP!]
[!ENDCODE!]
```

Without autospacing, the following code would be generated (to make spaces and newlines visible, they are printed as "." and "<CR>"):

```
..<CR>
....myoutput<CR>
..<CR>
....myoutput<CR>
..<CR>
<CR>
```

With activated auto spacing, the output would look like this:

```
....myoutput<CR>
....myoutput<CR>
```

5.12.10.5. Indentation

If you need the same indentation for several lines, you can use the INDENT-command which gets the amount of spaces used to indent as parameter. All output within an INDENT-block will be indented.

```
[!AUTOSPACING!]
[!INDENT "4"!]
  [!LOOP "text:split('1 2')"!]
    [!IF ".='1'"!]
      eins
    [!ELSE!]
```

```
        zwei
    [!ENDIF!]
  [!ENDLOOP!]
  drei
[!ENDINDENT!]
```

The template-code above produces the following well-formatted output:

```
    eins
    zwei
    drei
```

5.12.11. Excluding files from the generation

A file can be excluded from generation with a special tag. When the code generator encounters the **[!SKIPFILE!]** tag it marks the file. When the application comes across this marked file, it is not generated at all (although it is parsed completely issuing errors and warnings). The **[!SKIPFILE!]** tag comes in two shapes:

```
[!SKIPFILE!]

[!SKIPFILE "<XPath-Expression>"!]
```

In the first form the file is not generated in any case. In the second form the file is only generated if the XPath-Expression evaluates to false.

A code template can have multiple **[!SKIPFILE!]**-statements in which case the last one dominates all other statements.

5.12.12. Asserts, errors and warnings

To check the configuration during generation and issue errors and warnings to the user, the code generator supports four constructs: asserts, errors, warnings and info messages. Messages submitted via these constructs are handed to the application via a callback interface.

Errors and assertions stop the generation process immediately while warnings and info messages only submit a message to the application.

Errors, warnings and info messages all use the same syntax:

```
[!ERROR "<Message>"!]

[!ERROR!]
<Message>
[!ENDERROR!]


[!WARNING "<Message>"!]

[!WARNING!]
<Message>
[!ENDWARNING!]

[!INFO "<Message>"!]

[!INFO!]
<Message>
[!ENDINFO!]
```

Each construct comes in two flavors.

The first one directly includes the message in the starting tag. No end tag is needed (e.g. **[!ENDERROR!]** etc.). In this case the message is interpreted as plain text if it does not start with a question mark. If it starts with a question mark the rest of the message is evaluated as an XPath expression.

The second form of the construct is more powerful than the first one. In the second form the message can contain all construct described so far (**[!IF!]**, **[!LOOP!]**...).

The assert statement is a special form of the error statement. It includes an XPath expression. Only if this expression evaluates to false the message is printed and the generator run ends. Like the other constructs the assert comes in two shapes:

```
[!ASSERT "<XPath expression>","<Message>"!]

[!ASSERT "<XPath expression>"!]
<Message>
[!ENDASSERT!]
```

The two shapes work the same way as the other constructs.

5.13. The EPC file generator

The EPC File Generator is useful for all developers using external generators. To invoke external generators usually `.epc` files are necessary.

EB tresos Studio uses `.xdm` files to store the configuration. To be able to use the external generators in combination with EB tresos Studio , an additional generator is provided: The EPC File Generator.

The EPC File Generator is easy to use:

- Use the EPC File Generator in the generator extension-point for all modules that provide an external generator.
- Run the generators in generate mode from the GUI or commandline (project-based mode or legacy mode) to create required `.epc` files in the output directory.

The EPC File Generator can be registered in the `plugin.xml` of the respective module via the generator extension-point:

```
<extension point="dreisoft.tresos.launcher2.plugin.generator"
          id="EPCGenerator"
          name="EPC Generator">
 <generator moduleId="Lin_TS_T17D12M1I1R0"
            id="Lin_TS_T17D12M1I1R0GeneratorId"
            class="dreisoft.tresos.autosar2.generator.EPCFileGenerator">
   <parameter name="cfgFilePath" value="output"/>
   <parameter name="generateAllModules" value="true"/>
   <parameter name="generateIntoOneFile" value="false"/>
   <parameter name="contentType" value="asc:2.1"/>
 </generator>
</extension>
```

Note

To be able to access the EPCFileGenerator the `MANIFEST.MF` of the module must define the plug-in `dreisoft.tresos.autosar2.plugin` as a required bundle:

```
Require-Bundle: dreisoft.tresos.autosar2.plugin
```

Available parameters for the ECU File Generator are:

cfgFilePath
: path relative to the output directory in which to create the `.epc` files.

generateAllModules
: This is a boolean parameter that tells the EPC File Generator if all module configurations of the processed project must be exported or only the module configuration of the module for which the generator is registered. The default value is "true". If this parameter is set to "false", the parameter "generateIntoOneFile" is ignored. The generated module configuration will always be stored into a file called `<Module-Type>.epc`.

generateIntoOneFile
: specifies if all module configurations is to be stored into one single file which is always called `project.epc` or if each module configuration is to be stored separately into one file which is called `<Module-Type>.epc`. The default value is "false". If each module is exported into a single file only the first module configuration is exported.

contentType
: specifies in which content type the files should be generated. The default value is "asc:2.0". For all supported content types see EB tresos Studio UsersGuide Appendix: "Support for the Autosar Format".

Example: A project with 3 module configurations:

- Com (with generator extension point for EPCFileGenerator; generateAllModules="true"; generateIntoOneFile="true"; cfgFilePath="output")
- EcuC
- Lin (with generator extension point for EPCFileGenerator; generateAllModules="true"; generateIntoOneFile="false"; cfgFilePath="output")

If the user runs "generate" from the Commandline or the GUI several files will be generated into the folder "output" relative to the output directory. First all module configurations will be generated into one file: `project.epc`. And additionally 3 single files will be generated in addition: `Com.epc`, `EcuC.epc` and `Lin.epc`.

5.14. Using external generators

This chapter describes how external generators (provided as `.exe` or `.bat` can be integrated into and then be called either from the command line or the GUI.

5.14.1. Features

- Operate in project as well as in legacy mode.

- Export the project configuration in the form of AUTOSAR ECU Configuration Description files for all supported AUTOSAR versions.
- Run external tools (`.exe`-files) as code generator from GUI and command line.
- Run external batch-scripts (`.bat`-files) as code generator from GUI and command line.
- Parse generator output and feed it back to error logging facilities.

5.14.2. Functionality

The External Generator Engine first exports the project configuration or parts of the configuration to one or multiple AUTOSAR configuration files beneath the generation path (output directory). After that the actual external generator is run. The output of the generator is parsed back to .

5.14.3. Prerequisites

External generators are bound to a module like the normal template based code generators.

Therefore a normal module plug-in that will host the generator must be available. The module must include:

- module registration in the `plugin.xml`
- XDM schema

5.14.4. Adding the Generator to the plug-in

The generator executable itself must be copied to the module plug-in for which it generates code. To do that a directory `bin/` should be added to the module plug-in and the executable including necessary companion files should be added to this directory resulting in a file structure comparable to:

```
Can_TS_T16D4M2I0R0/META-INF/MANIFEST.MF
Can_TS_T16D4M2I0R0/plugin.xml
Can_TS_T16D4M2I0R0/config/Can.xdm
Can_TS_T16D4M2I0R0/bin/Cangen.exe
Can_TS_T16D4M2I0R0/src/...
Can_TS_T16D4M2I0R0/include/...
```

If the generator needs `.dll` that are not installed to the Windows system directory these `.dll` files could be copied to `lib/` beneath the tresos base installation directory.

5.14.5. Registering an External Generator

The generator must be registered via the generator Eclipse extension-point in the `plugin.xml` of the corresponding plug-in.

```
<extension point="dreisoft.tresos.launcher2.plugin.generator"
        id="Generator_Det_TS_T16D4M2I0R0"
        modes="myMode"
        name="Det TemplateGenerator">
   <generator id="ExtGenDet"
              moduleId="Det_TS_T16D4M2I0R0"
              class="dreisoft.tresos.autosar2.generator.ExternalGenerator">
       <parameter name="cfgFilePath" value="epc"/>
       <parameter name="generateAllModules" value="true"/>
       <parameter name="generateIntoOneFile" value="true"/>
       <parameter name="contentType" value="asc:2.1"/>
       <parameter name="shell" value="false"/>
       <parameter name="cwd" value="${pluginPath}/generator"/>
       <parameter name="commandline"
                  value="det.exe -f ${configFiles} -o ${outputDir}"/>
       <parameter name="error" value="^Error:\s*$m\s*$"/>
       <parameter name="warning" value="^Warning:\s*$m\s*$"/>
       <parameter name="info" value="^Info:\s*$m\s*$"/>
       <parameter name="evalExitCode" value="false"/>
   </generator>
</extension>
```

The External Generator interface that runs the generator is implemented in the class `dreisoft.tresos.autosar2.generator.ExternalGenerator`.

With the optional xml attribute `modes`, you may define generation modes. If they are not defined, "generate,verify" is used as default instead. You may trigger the mode(s) via the GUI or from the command line. For instructions how to do that, refer to Section 4.12.3.6: Running custom-built generation modes for legacy modules.

How the generator is run is defined by the parameters to the registration. The following parameters are available:

Parameter	Default	Description
cfgFilePath		The parameter `cfgFilePath` selects the path in which configuration file are generated. This path is interpreted relative to the generation path.
generateAllModules	false	If the parameter `generateAllModules` is set to `true` all configurations for all modules that are enabled in the project are generated.
generateIntoOneFile	false	If the parameter `generateIntoOneFile` is set to `true` one single configuration file is generated with the name `project.epc` and which contains all generated configurations.
contentType	asc	The parameter `contentType` sets the AUTOSAR ECU Configuration version with which the configuration is generated to disk. All available version (`asc`, `asc:2.0`, `asc:2.1`, `asc:3.0`...) are usable here.
commandline		The parameter `commandline` is mandatory and stores the commandline that calls the generator.
error		The parameter `error` contains the grammar via which to parse errors out of the output of the generator.
warning		The parameter `warning` contains the grammar via which to parse warnings out of the output of the generator.
info		The parameter `info` contains the grammar via which to parse info messages out of the output of the generator.
shell	false	The parameter `shell` must be set to `true` if the generator is implemented as a batch-file.
cwd		The parameter `cwd` can be used to specify the working directory from which to run the generator.
timeout		The parameter `timeout`, when given, gives a number of seconds after which the generator will be killed if it did not finish yet.
evalExitCode	false	The optional parameter `evalExitCode` must be set to `true` if the exit code of the executed commandline shall be evaluated. An exit code different from 0 (zero) will be interpreted as an error in that case. If set to false, the exit code will be ignored.

Table 5.46. External Generator Parameters

5.14.6. Generator Commandline

The commandline defined by the parameter `commandline` and the working directory, with which the generator is started, defined by `cwd` may contain variables which are expanded right before the parameter is used by the External Generator Engine. Variables always have the form:

```
${variable-name}
```

As an example the commandline of the generator call could have been set with:

```
<parameter name="commandline"
           value="${pluginPath}/bin/Cangen.exe -f ${configFiles} -o
           ${outputDir}/can"/>
```

The following variables are expanded:

Variable	Description
outputDir	The variable `outputDir` contains the generation path.
generationDir	The variable `generationDir` contains the generation path as stored in the project settings. In legacy mode it be the same as the variable `outputDir`.
tresosBase	The variable `tresosBase` stores the absolute path to where is installed.
pluginPath	The variable `pluginPath` stores the absolute path to where the Eclipse plug-in is installed that registers the generator.
variant	The variable `variant` stores the EB tresos Studio variant string (e.g. 2008.a).
buildnr	The variable `buildnr` stores the EB tresos Studio buildnr.
relversion	The variable `relversion` stores the EB tresos Studio release version.
project	The variable `project` stores the name of the project that will be generated. In legacy mode the project name is given with the system property `project`.
module	The variable `module` stores the module-id of the module that will be generated.
modules	The variable `modules` stores a space separated list of module-ids of all modules in the project that will be generated. In legacy mode these are the names of all module-ids that have been loaded.
generatedModules	The variable `generateModules` stores a space separated list of module-ids of all modules in the project that are generating code.
enabledModules	The variable `enabledModules` stores a space separated list of module-ids of all modules in the project that are enabled.
configFiles	The variable `configFiles` stores the a space separated list of the config files that were generated as absolute paths.
defaultConfig	The variable `defaultConfig` stores the name of the first configuration for the module that will be generated.
configs	The variable `configs` stores a space separated list of the names of all configurations for the module that is been generated.
generateConfigs	The variable `generateConfigs` stores a space separated list of the names of all configurations that should be generated for the module that will be generated.
enabledConfigs	The variable `enabledConfigs` stores a space separated list of names of all configurations that are enabled for the module that will be generated.
moduleSpecVersion	The variable `moduleSpecVersion` stores the full spec version of the module that will be generated as extractable from the module registration of the module.
moduleSoftwareVersion	The variable `moduleSoftwareVersion` stores the full software version of the module that will be generated as extractable from the module registration of the module.

Variable	Description
moduleReleaseVersion	The variable `moduleReleaseVersion` stores the full release version of the module that will be generated as extractable from the module registration of the module.
moduleType	The variable `moduleType` stores the type of the module that will be generated as extractable from the module registration of the module.
moduleCategory	The variable `moduleCategory` stores the category of the module that will be generated as extractable from the module registration of the module.
moduleLayer	The variable `moduleLayer` stores the layer of the module that will be generated as extractable from the module registration of the module.
target	The variable `target` stores the target of the project that will be generated. In legacy mode the target can be set with the system property `target`.
derivate	The variable `derivate` stores the derivate of the project that will be generated. In legacy mode the derivate can be set with the system property `derivate`.
ecuid	The variable `ecuid` stores the ecuid of the project that will be generated. In legacy mode the ecuid can be set with the system property `ecuid`.
mode	The variable `mode` stores the mode of the generator that is running.

Table 5.47. Variables for commandline and cwd

5.14.7. Parsing the Output of the Generator

To parse back the output of the generator the grammar of the output must be specified. The External Generator Engine then parses standard output and standard error output of the generator and creates log entries for the error logging facility of . If any errors are logged the generation is assumed to be failed.

The grammar for parsing back the output is specified with the parameters:

- error
- warning
- info

All three parameters are optional. The grammar for error messages is specified with the parameter `error`, the one for warnings with the `warning` parameter and the one for info messages with the `info` parameter.

The grammar has the following form (EBNF):

```
( variable | regexp )*
```

The grammar is composed of (perl) regular expressions and variables. Variables have the form:

```
"$" variableName
```

where variable name is a single character. Variables match the part between the regular expressions. If a variable is given as the first token in the grammar it matches the text between the start of the yet unparsed output and the first regular expression. If a variable is given as the last token of a grammar it matches the end of the output.

The grammar allows to parse multi line messages by using the special variable `$L`. The $L variable is actually not a real variable but serves as a marker to mark the beginning of a continued line pattern. Everything before the first $L must match the first line of a message printed by the generator. Afterwards the parser tries to match the continued line patterns. A continued line pattern is the grammar between a $L variable and another $L variable or a $L variable and the $N variable or a $L variable and the end of the grammar.

The grammar also allows to define multiple alternative patterns for one message type (error, warning, info). An alternative pattern is introduced by the marker variable `$N`. Everything from a $N to the next $N or the end of the pattern is considered as an alternative grammar pattern. Each such pattern can contain continued line patterns whereas the first pattern is always the first line pattern.

Consider the following example:

```
<parameter name="error"
           value="ERROR:$m$LFILE:$f$LLINE:$NFATAL:$m$LFILE:$f$LLINE:$l"/>
```

This pattern matches error messages of the following forms:

```
ERROR: error message
FILE: test.c
LINE: 10

FATAL: fatal error message
LINE: 5
FILE: test.c

ERROR: error message
FILE: test.c

FATAL: fatal error
LINE: 7
```

Any line that does not match a pattern is ignored.

All other variables are used to fill in parts of the messages returned to . The following variables can be used:

Variable	Description
$f	the variable `$f` matches the filename printed in the error message.
$l	the variable `$l` matches the line printed in the error message.
$m	the variable `$m` provides the error message.

Table 5.48. Variables for commandline and cwd

The resulting message printed by looks either the following way if at least file name message variables are used:

```
"[" filename ":" line "]" message
```

otherwise only the message is printed.

5.15. ECU Resource Manager

EB tresos Studio provides a feature called ECU Resource Manager. The ECU Resource Manager allows to register multiple values for ECU resources like RAM, ROM, etc. Each resource value is bound to an ECU type. The type is primarily identified by target and derivate. Additionally, it may depend on a configured module or arbitrary configuration values, e.g. a subderivate identifier.

The ECU Resource Manager provides XPath functions to query these resource values. So it can be used by code templates or XDMs that apply to several derivates or subderivates of the same target but require specific configuration values in detail.

Tip

A complete example of how to use the ECU Resource Manager can be found in the demo plug-in *PublicApiTest_TS_T00D0M0I0R0* which is located in *<tresos-install-dir>/demos*. This demo plug-in can be imported to your eclipse installation by right-clicking in the Package Explorer and select *Import... -> Import Existing Project Into Workspace*

5.15.1. Registering ECU Resource Properties

ECU resources are described in the form of Java properties files. These files store key/value pairs in the following form:

```
<key>:<value>
```

The key must be unique. To create multi-line values, quote the line end with a trailing backslash. Leading white space of continued lines is ignored. Comments can be introduced by placing a '#' at the beginning of a line.

A snippet from a resource properties file could look like this:

```
# ECU resources for V850EGP1
Ram:512
Flash:2048
Rom:128
Port1.Func:Can,Dio
```

An ECU resource properties file must be placed inside an Eclipse plug-in (usually a module plug-in) that is installed into the `plugins` directory of EB tresos Studio.

To enable the ECU Resource Manager for a plug-in the `dreisoft.tresos.autosar2.api` must be added to the list of required plug-ins. Therefore change the `META-INF/MANIFEST.MF` of the plug-in as follows:

```
Manifest-Version: 1.0
...
Require-Bundle: dreisoft.tresos.autosar2.api,
...
```

The ECU resource properties file must be registered at the extension point `dreisoft.tresos.autosar2.api.plugin.ecuresource` in the `plugin.xml`. The `target` and `derivate` attributes are mandatory and used to filter the list of registered properties files to the current project respectively the module configurations. The mandatory `file` attribute defines path to the properties file relative to the plug-in's root folder.

The optional attribute `moduleId` defines the module for which these properties are mainly registered for. Normally this will be the module defined by the same plug-in.

The `moduleId` attribute was first introduced in EB tresos Studio 2009.a and is recommended to use from that version on. When a module is defined it doesn't mean that these ECU resource properties are only available for that module. But these properties are only available if the current project respectively the module configurations contain at least one module configuration of the module defined by this attribute.

Note

Before EB tresos Studio 2009.a it was recommended to register exactly one ECU resource properties file that contains information for any module of the same target and derivate because the first relevant properties file was returned on a request and any other registered properties file was not available. Therefore, it was recommended to deliver the properties file within a single *resource* plug-in that was unique for one target/derivate.

With introducing the `moduleId` attribute in EB tresos Studio 2009.a also a merge strategy for ECU resource properties files has been implemented. Any properties that are registered using the new `moduleId` attribute are merged on any request. It is therefore recommended to deliver an ECU resource properties file, which contains only module-specific data, for each module that requires ECU resources. Please make sure that the key of an ECU resource property is still unique after merging, e.g. by prefixing the keys with the module's ID.

To stay downward-compatible, older registrations that do not define a module are not merged and behave in the old way. Nevertheless, if both registrations with and without attribute `moduleId` are found, then the registrations with `moduleId` are prioritized.

The following example shows a typical ECU resource properties registration for `target` V850 and `derivate` V850EGP1 for module `Can` in the `plugin.xml` of the module's plug-in:

```
<extension point="dreisoft.tresos.autosar2.api.plugin.ecuresource">
  <ecuresource
          target="V850"
          derivate="V850EGP1"
          moduleId="Can_TS_T16D4M2I1R0"
          file="resources/V850_V850EGP1.properties">
  </ecuresource>
</extension>
```

5.15.1.1. Advanced properties

The extension point `dreisoft.tresos.autosar2.api.plugin.ecuresource` supports optional `parameter` tags to add more search criteria to an ECU resource properties file than the mandatory `target` and `derivate`. These parameters must correspond to parameters of registered ECU Resource Finders. Please read more about ECU Resource Finders below.

```
<extension point="dreisoft.tresos.autosar2.api.plugin.ecuresource">
  <ecuresource
          target="V850"
          derivate="V850EGP1"
          moduleId="Can_TS_T16D4M2I1R0"
          file="resources/V850_V850EGP1.properties">
      <parameter name="subderivate" value="ERAY" />
  </ecuresource>
</extension>
```

5.15.2. ECU Resource Finder

It is not intended to access an ECU resource properties file directly. The ECU Resource Manager uses so called ECU Resource Finders to resolve the relevant resource properties files for the configuration that wants to access a resource property. An ECU Resource Finder has to implement interface `dreisoft.tresos.autosar2.api.ecuresource.IEcuResourceFinder`. It neither depends on a special resource properties file nor on the plugin that registered it.

It is recommended that each plugin that registers an ECU resource properties file also registers an ECU Resource Finder at extension point `dreisoft.tresos.autosar2.api.plugin.ecuresourcefinder`. You may write your own implementation of interface `IEcuResourceFinder`, but the functionality provided by the default

implementation `dreisoft.tresos.autosar2.api.ecuresource.DefaultEcuResourceFinder` works for most cases. The following snippet registers a `DefaultEcuResourceFinder` at the `plugin.xml` of the plugin that wants to use the ECU Resource Manager:

```
<extension point="dreisoft.tresos.autosar2.api.plugin.ecuresourcefinder">
  <ecuresourcefinder>
    <finder
class="dreisoft.tresos.autosar2.api.ecuresource.DefaultEcuResourceFinder"/>
  </ecuresourcefinder>
</extension>
```

5.15.2.1. Advanced properties

An ECU Resource Finder neither depends on an ECU resource properties file nor on a specific plugin or target (see Section 5.15.2: ECU Resource Finder for detailed information). Therefore any registered ECU Resource Finder is able to find any registered ECU resource properties file. Thus it is essential to the implementation of the interface `IEcuResourceFinder` to restrict the internal logic as much as possible to stay out of the way of other finders.

For this reason the extension point `dreisoft.tresos.autosar2.api.plugin.ecuresourcefinder` supports optional `parameter` tags, similar to the ECU resource properties extension. These parameters help to define further details for retrieving the relevant properties files than just the mandatory ones. Mandatory details are: `target` and `derivate`. If a `moduleId` is additionally defined then the properties are only relevant for configurations that contain a module configuration of that module.

The `DefaultEcuResourceFinder` only accepts parameter names starting with `name` (the name-parameter) and `path` (the path-parameter); both end with an arbitrary suffix. For any name-parameter one path-parameter must exist, so that one name-parameter and one path-parameter make a pair. You must not define a parameter twice within the same finder registration.

The value of the path-parameter must be a schema path. This schema path will be evaluated by the current configuration upon any request to the finder.

5.15.2.2. Default ECU Resource Finder Implementation

The `DefaultEcuResourceFinder` implementation is very restrictive. For a better understanding, which resource is returned on a particular request, it is good to know about the internal logic. This internal logic is explained in the following:

First of all the registered ECU resource properties files are filtered by `target`, `derivate` and `moduleId`. The requested values are automatically taken from the current project respectively the module configurations that form the context of the caller.

The number of name- and path-parameter pairs of a `DefaultEcuResourceFinder` registration must match the number of optional parameters of the registered ECU resource properties extension. If the `DefaultEcuResourceFinder` is registered without parameters, it will only consider ECU resource properties files registered without parameters.

Then the `DefaultEcuResourceFinder` checks each name-/path-parameter pair.

For each name- and path-parameter pair the `DefaultEcuResourceFinder` checks if a parameter exists for the registered ECU resource properties file with the value of the name-parameter as name.

If a corresponding parameter exists in the ECU resource properties extension for all name- and path-parameter pairs, then the values for these parameters are verified. The `DefaultEcuResourceFinder` retrieves the configuration value for the specified schema path of the path-parameter. The `DefaultEcuResourceFinder` compares this value with the value of the corresponding parameter value of the ECU resource properties extension. If all values are equal to the values of the current configuration, the ECU resource properties file matches the requested criteria.

Note

The properties are returned to the caller, only if the number and values of the parameters associated with an ECU resource properties file match the number and values of the name- and path-parameter pairs defined in the `DefaultEcuResourceFinder` registration.

```
  <extension point="dreisoft.tresos.autosar2.api.plugin.ecuresource">
    <ecuresource
      target="MB91"
      derivate="MB91460"
      moduleId="Com_TS_T19D1M2I1R0"
      file="resources/MB91_MB91461.properties">

      <parameter name="subderivate" value="MB91461" />
    </ecuresource>
  </extension>

  <extension point="dreisoft.tresos.autosar2.api.plugin.ecuresourcefinder">
    <ecuresourcefinder>
      <finder
class="dreisoft.tresos.autosar2.api.ecuresource.DefaultEcuResourceFinder">
        <parameter name="name" value="subderivate" />
        <parameter name="path"
```

```
value="/AUTOSAR/TOP-LEVEL-PACKAGES/TS_T/ELEMENTS/Base/SubDerivate" />
      </finder>
    </ecuresourcefinder>
  </extension>
```

The above registration fetches the subderivate from the configuration parameter `SubDerivate` of the module `Base` and compares it to the `subderivate` parameter of the ECU resource properties parameter `subderivate` shown above.

There are no limitations on the number of parameters in the ECU resource properties and finder extensions.

```
  <extension point="dreisoft.tresos.autosar2.api.plugin.ecuresource">
    <ecuresource
      target="MB91"
      derivate="MB91460"
      moduleId="Det_TS_T19D2M3I0R0"
      file="resources/MB91_MB91461.properties">

      <parameter name="subderivate" value="MB91462" />
      <parameter name="version" value="3.0" />
    </ecuresource>
  </extension>

  <extension
point="dreisoft.tresos.autosar2.api.plugin.ecuresourcefinder">
    <ecuresourcefinder>
      <finder
class="dreisoft.tresos.autosar2.api.ecuresource.DefaultEcuResourceFinder">
        <parameter name="nameDerivate" value="subderivate" />
        <parameter name="pathDerivate"
value="/AUTOSAR/TOP-LEVEL-PACKAGES/TS_T/ELEMENTS/Base/SubDerivate" />

        <parameter name="nameVersion" value="version" />
        <parameter name="pathVersion"
value="/AUTOSAR/TOP-LEVEL-PACKAGES/TS_T/ELEMENTS/Base/Version/SpecVersion" />
      </finder>
    </ecuresourcefinder>
  </extension>
```

5.15.3. ECU Resource XPath Functions

To access the content of ECU resource properties via XPath, use the following functions:

- `ecu:has(<key>)`: Checks if the found ECU resource properties contain a key with the given name, e.g. calling `ecu:has('Rom')` on the resource properties snippet above would return `true`.
- `ecu:get(<key>)`: Returns the value of the given key name in the found ECU resource properties or `null` if no value has been found, e.g. calling `ecu:get('Rom')` on the resource properties snippet above would return `128`.
- `ecu:list(<key>)`: Splits the value of the given key name at commas in the value and returns a node-set. Calling `ecu:list('Port1.Func')` on the ECU resource properties snippet above would return a node-set containing the 2 values `Can` and `Dio` (unlike `ecu:get('Port1.Func')` which would return the pure value `Can,Dio`).

The XPath function `ecu:parameters(String)` reads the values of the given parameter name between all registered ECU resource properties that match the target, derivate and optionally moduleId of the project (but not the optional parameters themselves). A common use-case for this function would be to use the return-values to fill a combo box with possible subderivates in the editor. To define further subderivates only requires you to register further ECU resource properties files using this approach.

```
ecu:parameters(<parameter-name>)
```

In the above example the following call

```
ecu:parameters('subderivate')
```

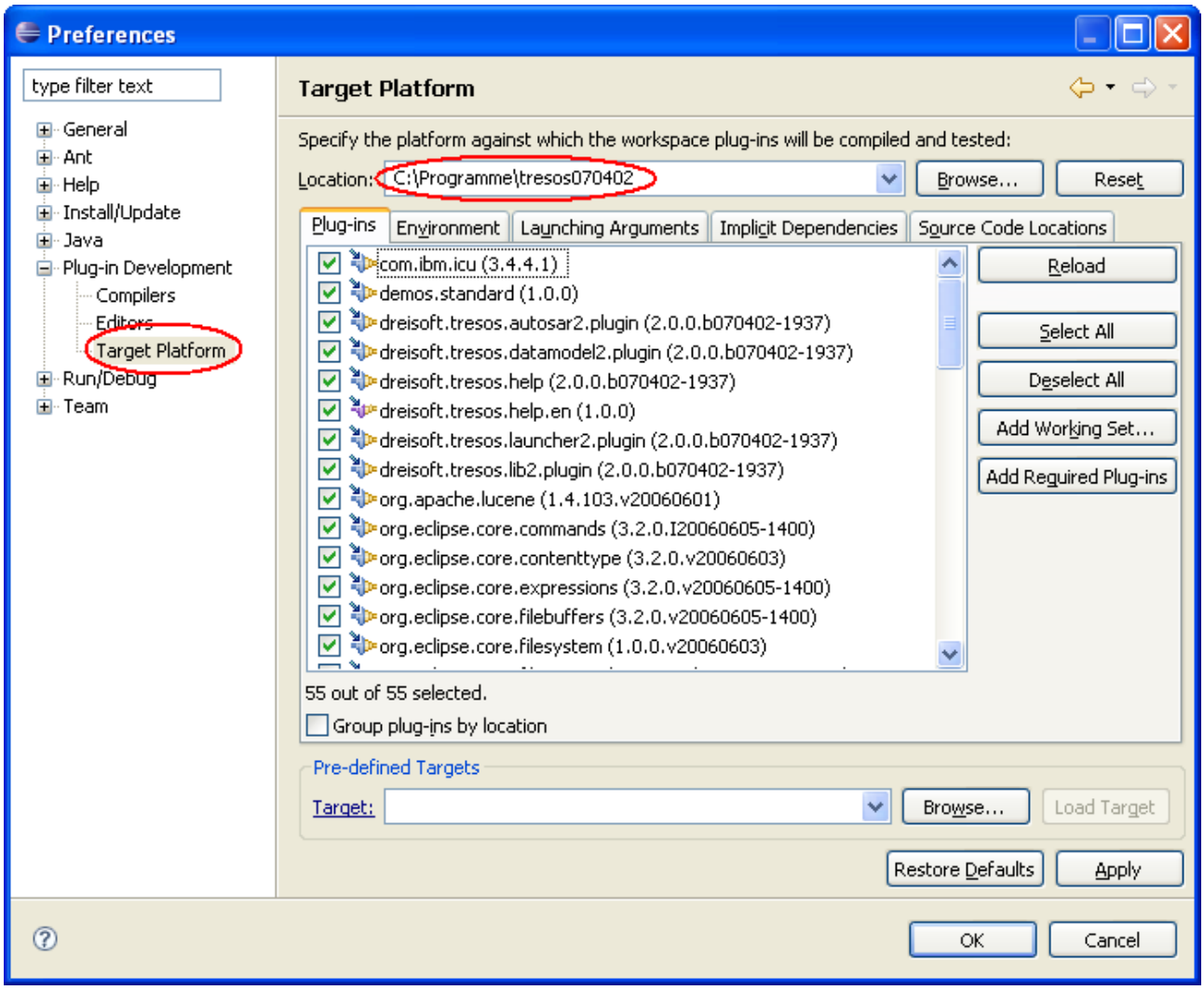
would return a list containing `MB91461` and `MB91462`.

5.16. Creating a module with Eclipse

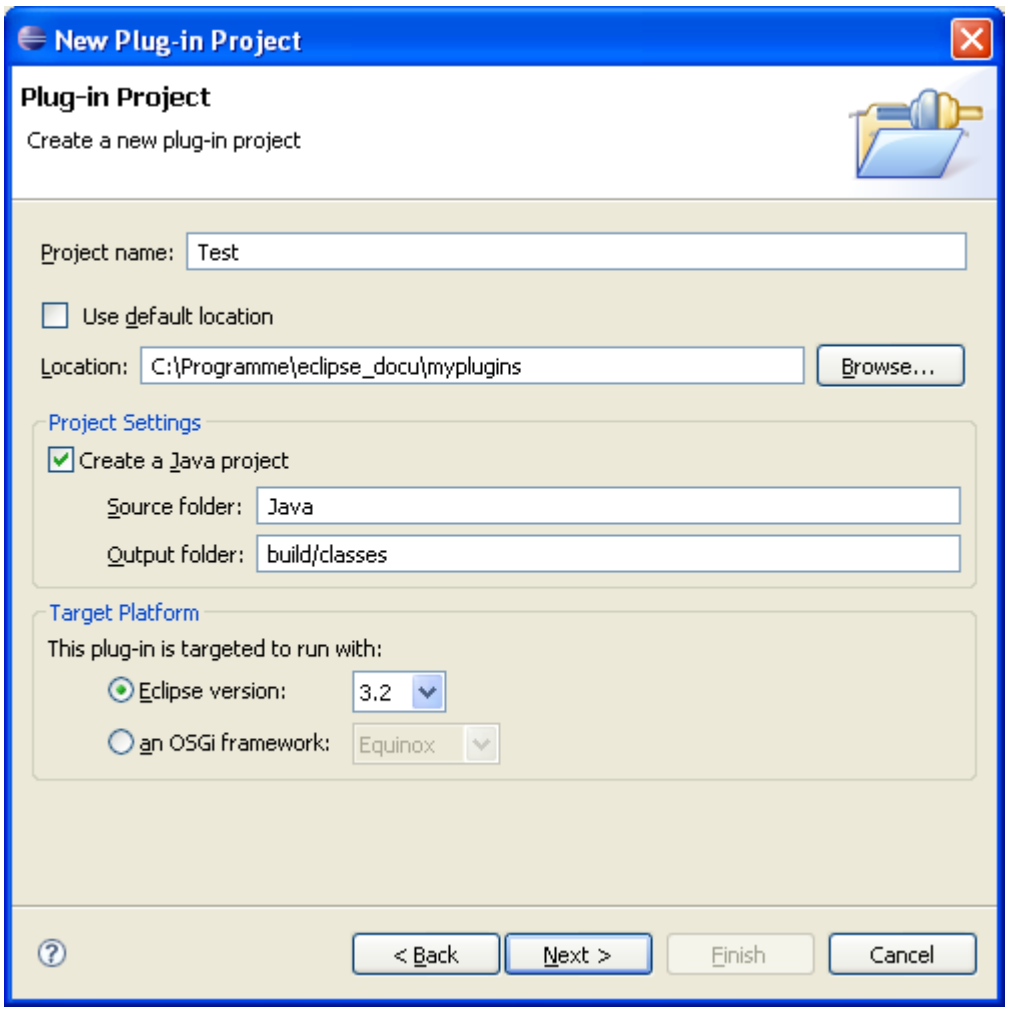
5.16.1. Installation and configuration

Assuming that you already installed EB tresos Studio, you first have to download the Eclipse SDK from eclipse.org and install it.

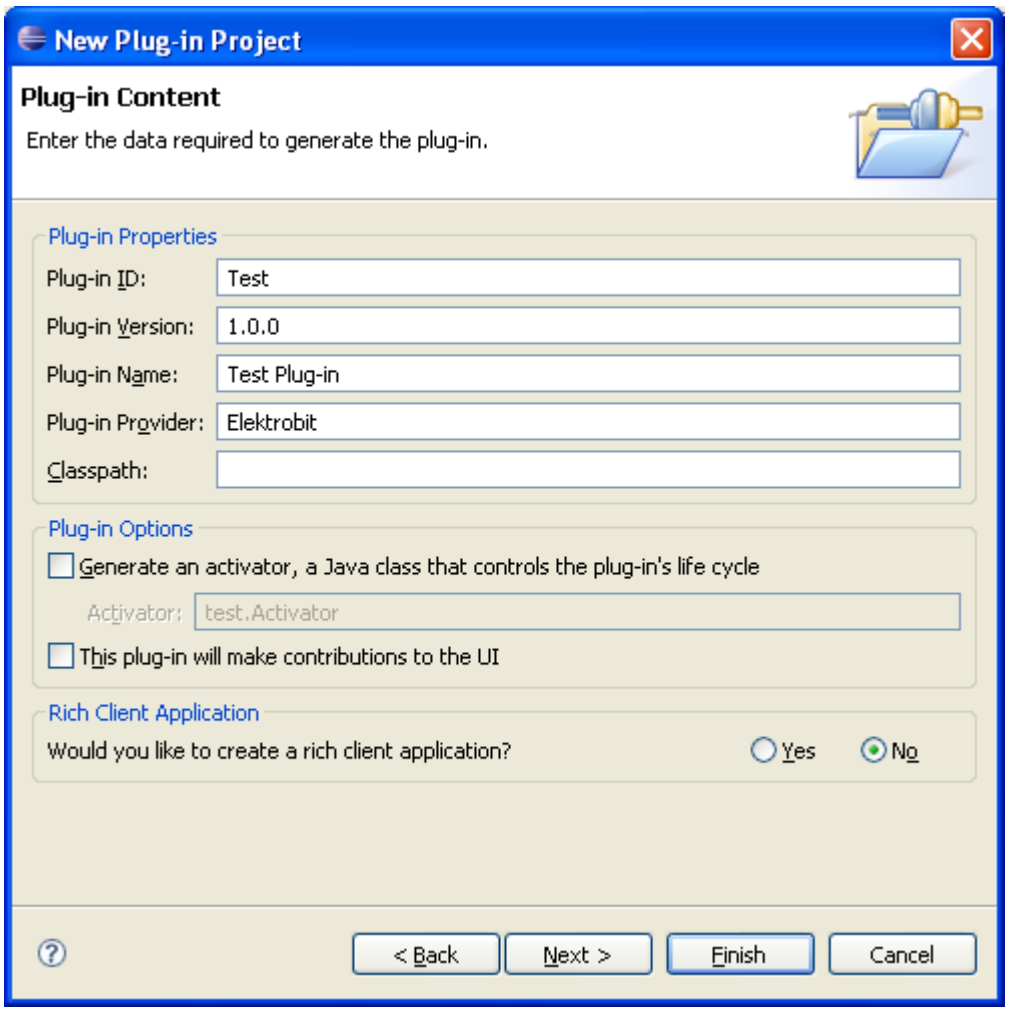
Start Eclipse and choose "Windows/Preferences..." and choose the base-directory of you EB tresos Studio installation as your target-platform.



Next, create a new Plug-in Project by selecting "File/New/Project..." and choose "Plug-in Project".

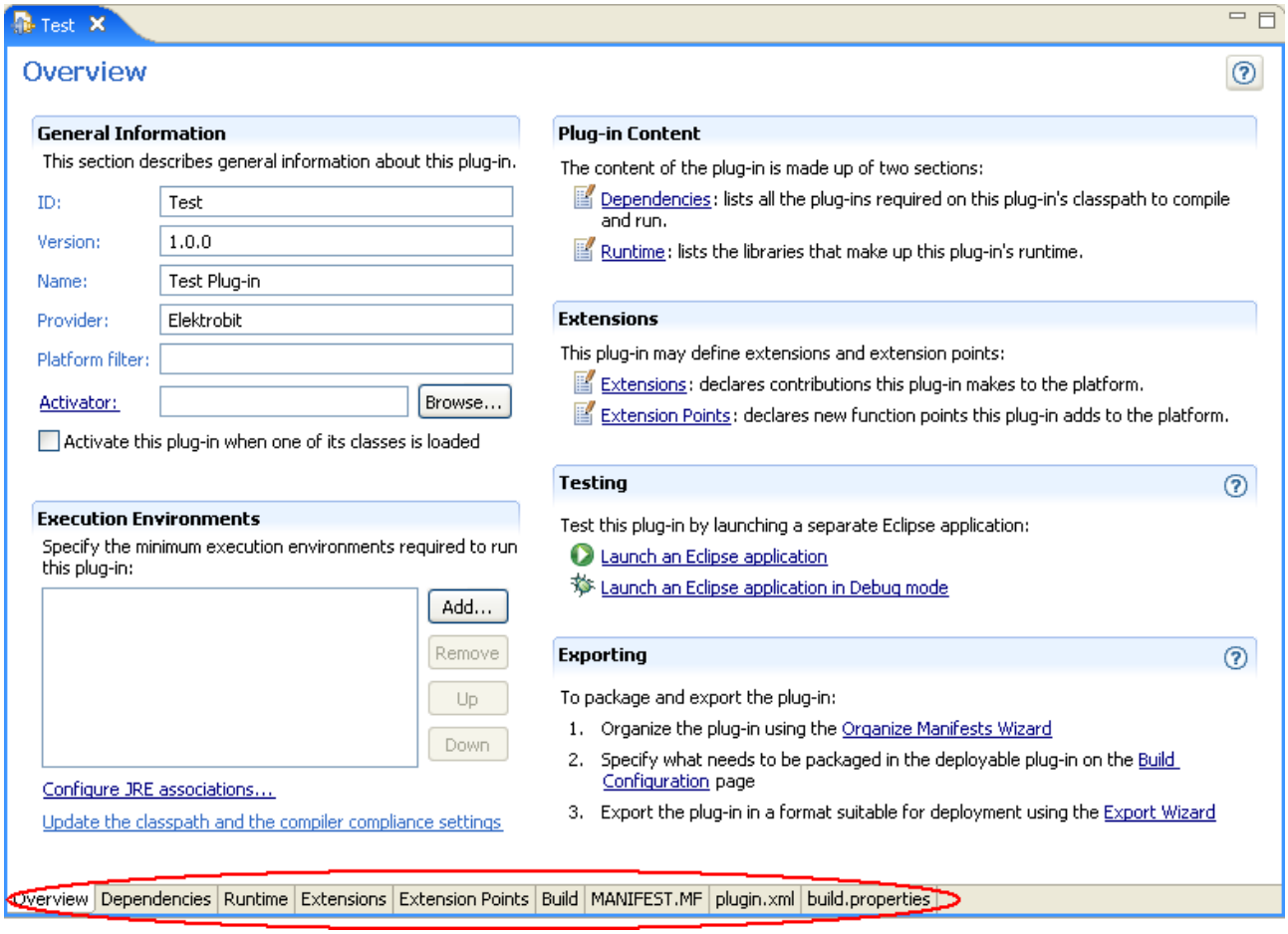


Choose the directory, where you want to create your new plugin. You only have to enable the checkbox "Create a Java project", if you want to develop Java-classes.

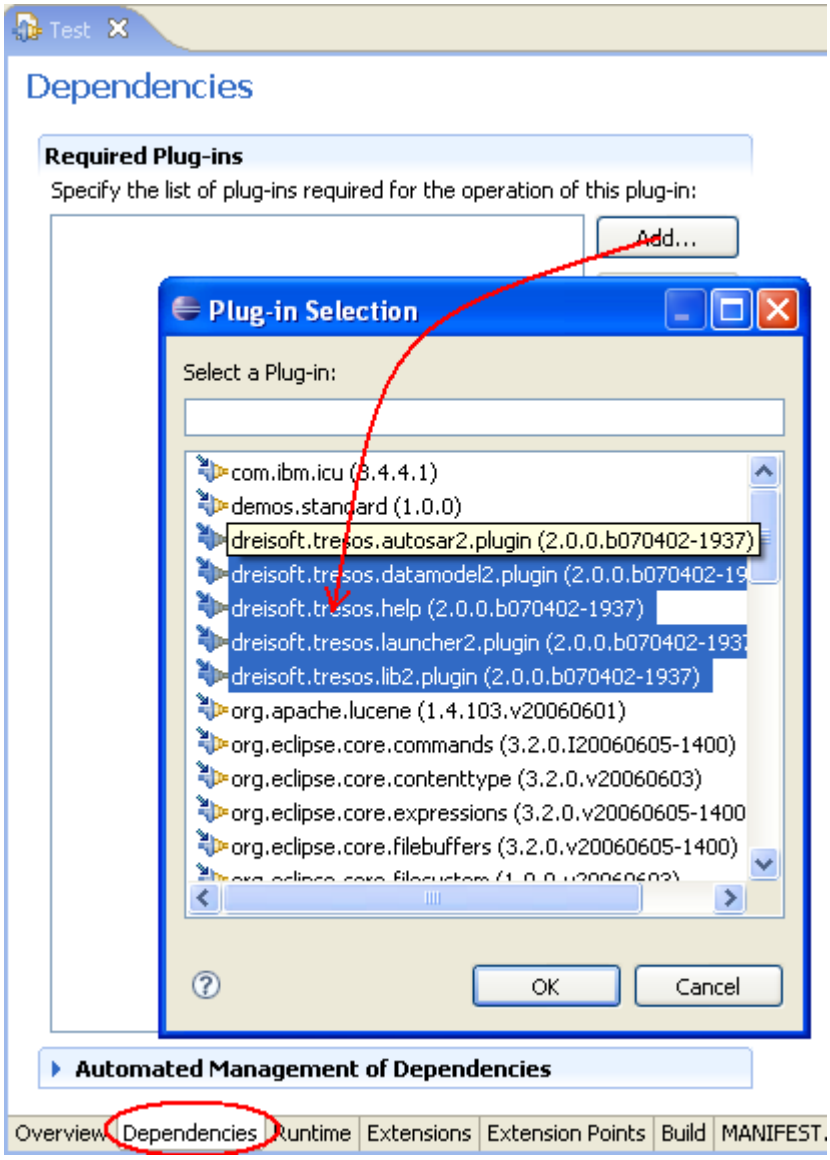


Deselect the checkboxes within the group "Plug-in Options" and click "Finish".

Within the PDE-editor, you can now change the configuration of your plugin and create extensions.



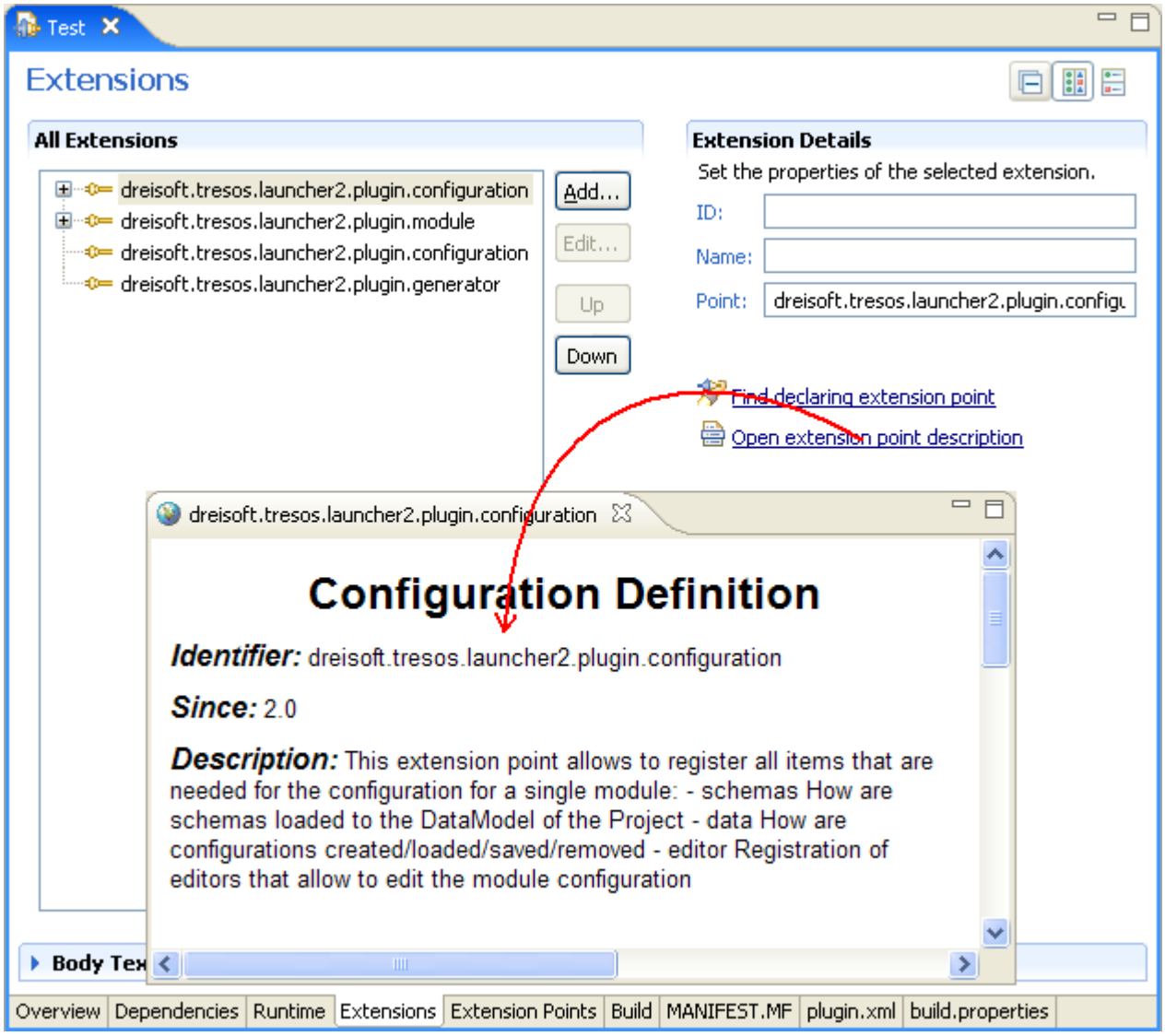
Use the tab-pages at the bottom of your editor to switch to the "Dependencies" configuration-page.



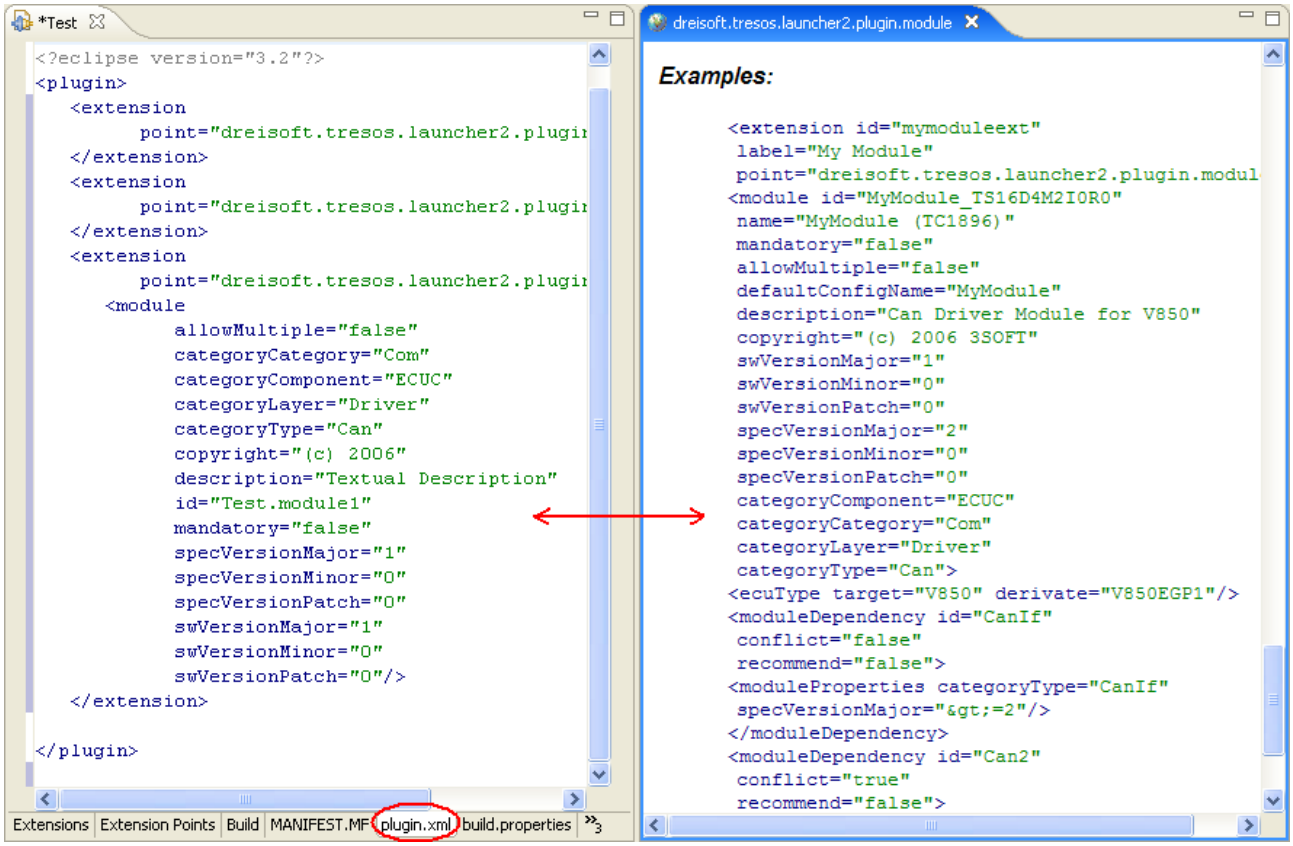
Add all dreisoft.* plugins to your dependencies. Now you can use all EB tresos Studio extension-points within your plugin.

5.16.2. Extension points and their description

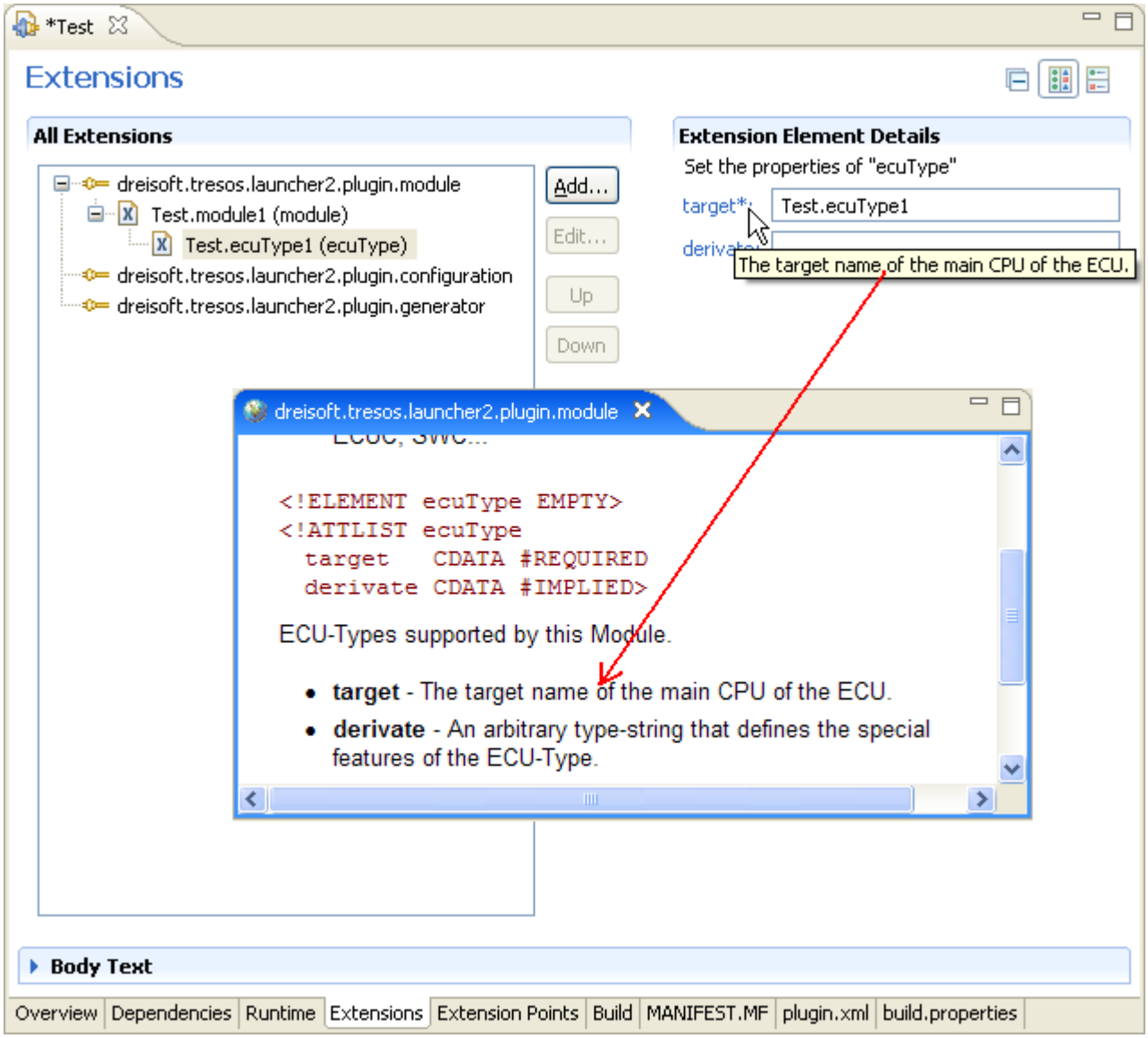
If you select an extension within the PDE-editor, you can click on the link "Open extension point description" to get a detailed description of all elements which belong to it.



The description-editor contains a description of the extension point and each of its elements which should be read first before using it.

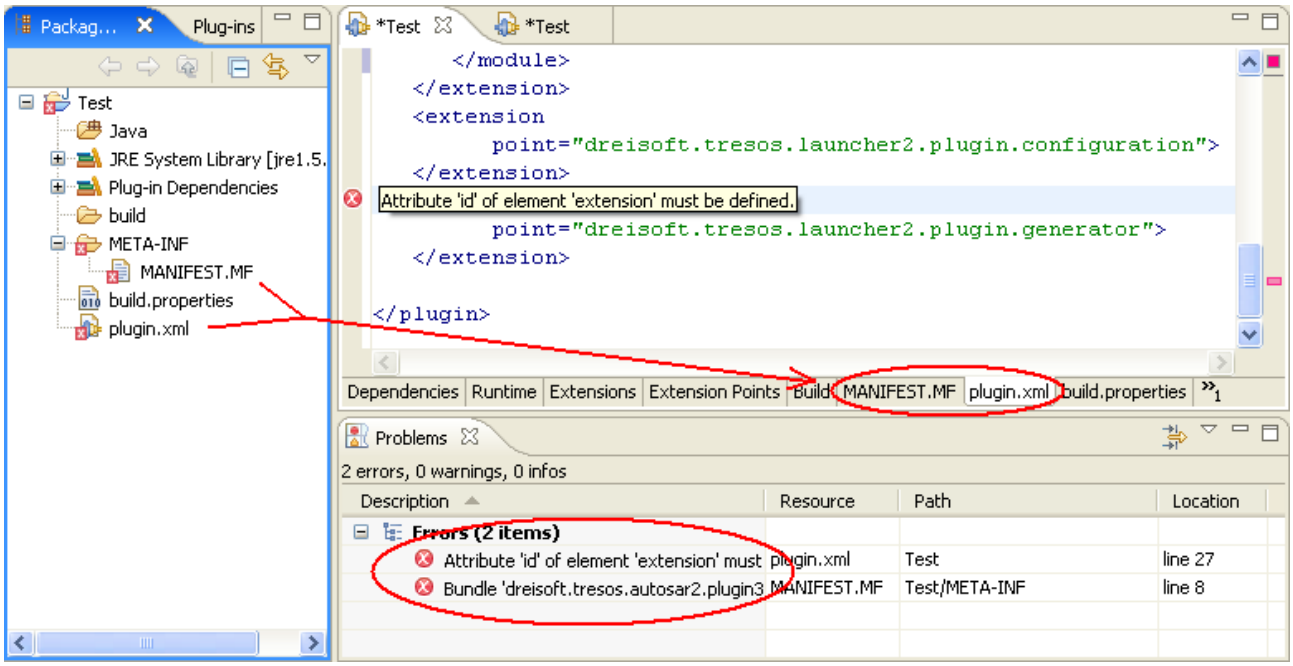


In most cases, there is an example at the bottom of the description-editor for each extension point. If you open the tab-page "plugin.xml" within your PDE-editor, you can compare your extension with this example, which is shown an xml-snippet.



Additionally, the tool tip of each element contains the description of each element which is a short version of the description shown in the description-editor.

5.16.3. Errors and warnings within the PDE



If there is an error within your plugin.xml or MANIFEST.MF-file, the file is marked with a red overlay-icon in the Package Explorer. Additionally, all errors are listed within the Problems-view (where you can click one to jump to the error). A description for each error is provided within the Problems-view and as tool tip in the editor.

5.16.4. Creating a EB tresos Studio ModulePlugIn

5.16.4.1. Schema-file and templates

First, you need your vendor specific module definition (VSMD) as xdm-file. Just put the file into a new directory "config" in your plugin.

Next, you need your code-templates, which you put into the directory "generate". You can put any directory-structure with any files into this directory - all of them will be generated, taking on your directory-structure.

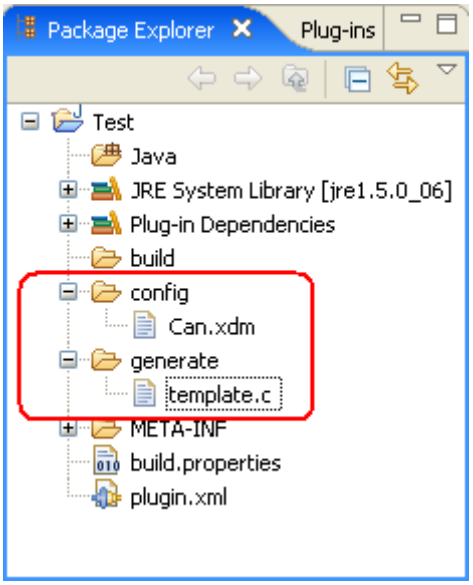


Figure 5.71. Files and folders

5.16.4.2. Extension points

Since there is a detailed description available for each element of the extension points, this chapter only describes the most interesting extension points and their elements. To create a new ModulePlugIn, you have to add at least the following three extensions to your plugin.

5.16.4.2.1. Extension point dreisoft.tresos.launcher2.plugin.module

This extension point defines the module of your plugin.

The most important element to configure is the ModuleId which will be referenced by the configuration- and generator-extension point.

Since the ecuType is mandatory, create at least one by choosing "New/ecuType" from the context menu of the module.

Figure 5.72. Module extension point

5.16.4.2.2. Extension point dreisoft.tresos.launcher2.plugin.configuration

This extension point defines your schema loader, dataloader and editor for your ModulePlugIn.

Element	Description
configuration	Needs the id of your module defined before (ModuleId)
schema/manager	must be dreisoft.tresos.autosar2.resourcehandling.AutosarSchemaManager
schema/resource	the value must be the relative path to your schema-file, the type must be "xdm"
data/manager	must be dreisoft.tresos.autosar2.resourcehandling.AutosarConfigManager
data/schemaNode	the SHORT-NAME-path to your MODULE-DEFINITION prefixed with "ASPath:"
editor/class	must be dreisoft.tresos.launcher2.editor.GenericConfigEditor
editor/parameter	name must be "schema", the value must be the SHORT-NAME-path to your MODULE-DEFINITION prefixed with "ASPath:"

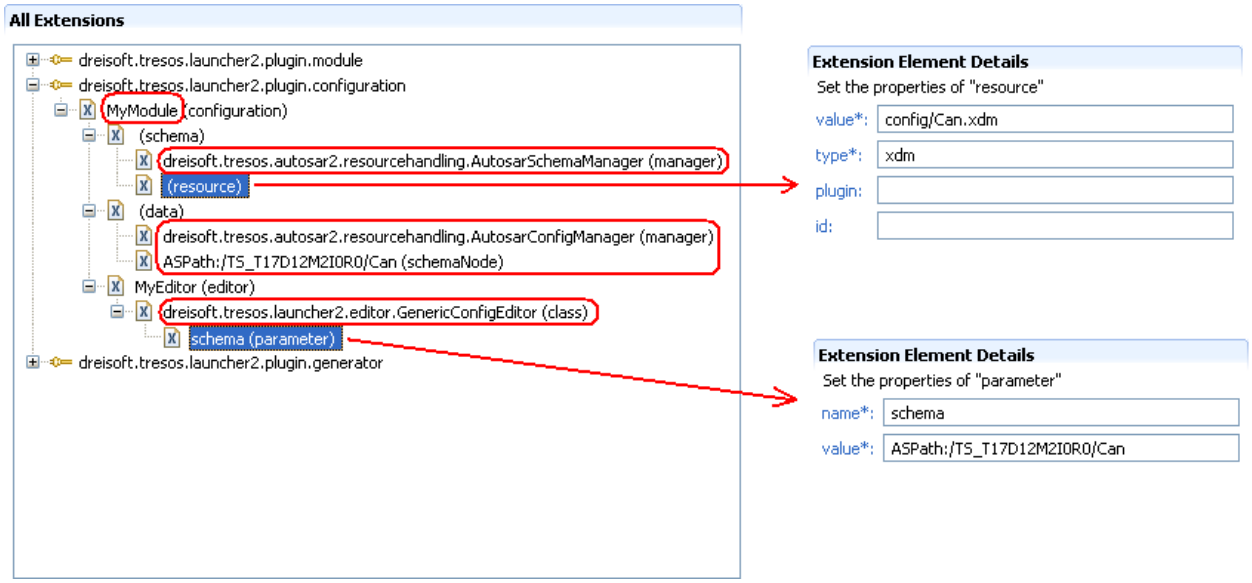


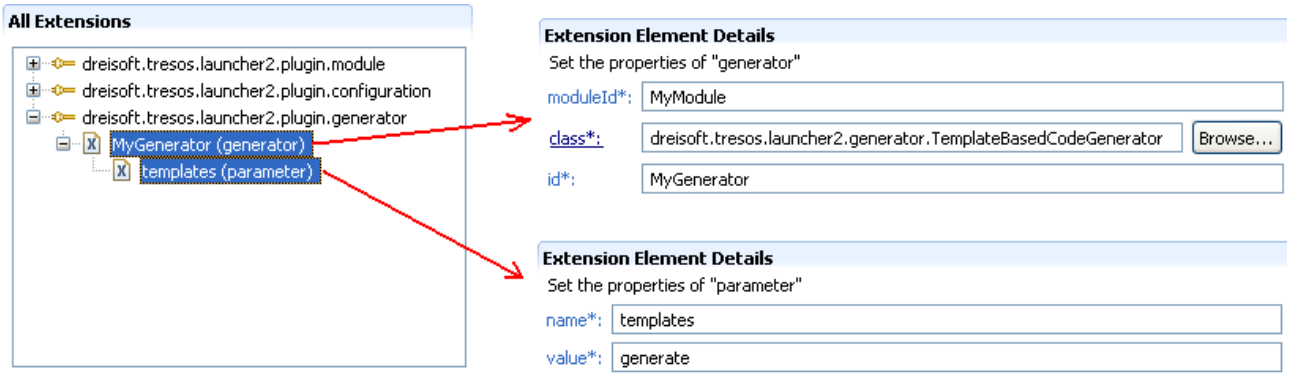
Figure 5.73. Module extension point

5.16.4.2.3. Extension point extension point dreisoft.tresos.launcher2.plugin.module

This extension point defines the generator which can be used for your ModulePlugIn.

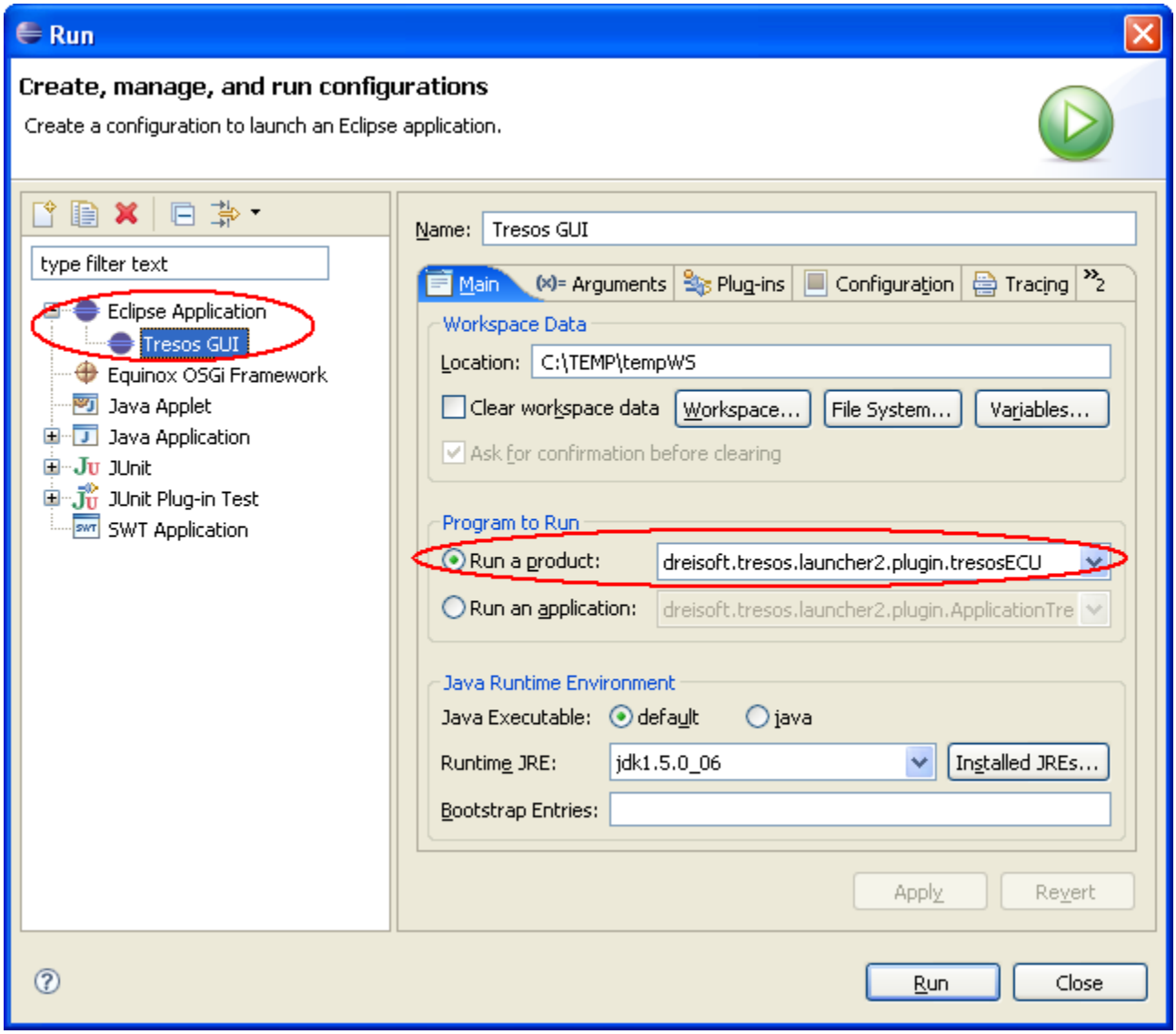
Again, you have to declare the id of your module (ModuleId) defined above. The class of your editor must be dreisoft.tresos.launcher2.generator.TemplateBasedCodeGenerator.

This generator needs the parameter "templates" with the relative path to your code-templates.

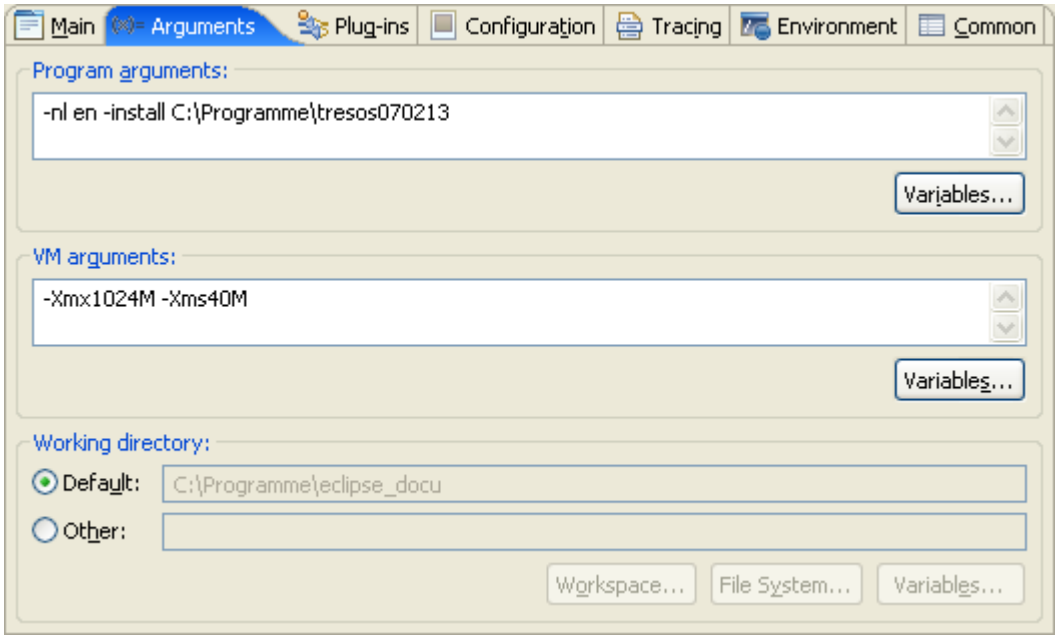


5.16.5. Starting EB tresos Studio from Eclipse

To start EB tresos Studio from within Eclipse, you have to create a new Run-configuration. For this, choose "Run/Run..." from the menu.



Create a new "Eclipse Application" and choose "dreisoft.tresos.launcher2.plugin.tresosECU" from the product-combobox as shown in the picture above.



On the Arguments-tab, add "-nl en -install <tresos-basedir>" to the Program arguments. The given tresos-basedir is only used to get the license from the contained bin-directory. Add "-Xmx1024M -Xms40M" to the "VM arguments" section.

Now, with a click on "Run", the EB tresos Studio configurator starts.

Caching-problems

Because eclipse is caching some things, it might be possible, that you cannot see changes made within your ModulePlugIn in the next run of EB tresos Studio To solve this problem, enable the checkbox "Clear workspace data" within the run-dialog.

5.17. The NG Generator Framework

With version 2008.b of EB tresos Studio a new generator framework is available. The so called new generation (NG) generator API brings a lot new features while supporting the classic template based and external code generators as well. Of course, the existing generator framework still works and there is no need to switch to the new API.

Note

This chapter gives only a short overview of the new generator framework. For detailed information and examples of the NG Generator API please refer to the API documentation of plugin `dreisoft.tresos.generator.ng.api.plugin`.

A complete demonstration can be found in the plugin *PublicApiTest_TS_T00D0M0I0R0* which is located in *<tresos-install-dir>/demos*. This demo plug-in can be imported into your Eclipse installation by right-clicking in the Package Explorer and select *Import... -> Import Existing Project Into Workspace*.

5.17.1. Generation Steps

The NG Generator introduces three generation steps for "pre" generation, "default" generation and "post" generation. The generator step will be specified by a new attribute "step" at the generator extension point. Each generator runs in exactly one step.

All generators scheduled to the "pre" generation step run before all generators scheduled to the "default" generation step. Pre-generation can be used to calculate data that is used by all code-generators or to prepare the output directory structure.

The "default" step is the main generation phase when code generators shall run. Thus the default step is not explicitly specified all existing generators are assumed to run in the default step.

All generators scheduled to the "post" generation step run after all generators scheduled to the "default" generation step. Post-generation can be used to generate code based on data that is gathered during the normal generator run like post-build structures (structure sizes) etc.

5.17.2. Module-Independent Generators

In addition to the new generation steps the NG Generator now supports module-independent generators. Those generators do not generate code for a specific module. This means that they always run no matter which modules are selected for code generation.

Module-independent generators can only be used for pre- and post- generation steps. In the pre-generation step they will run before all module-bound generators will run. And in the post-generation step they will run after all module-bound generators will run.

A module-independent generator is specified by omitting the attribute "moduleId" at the generator extension point.

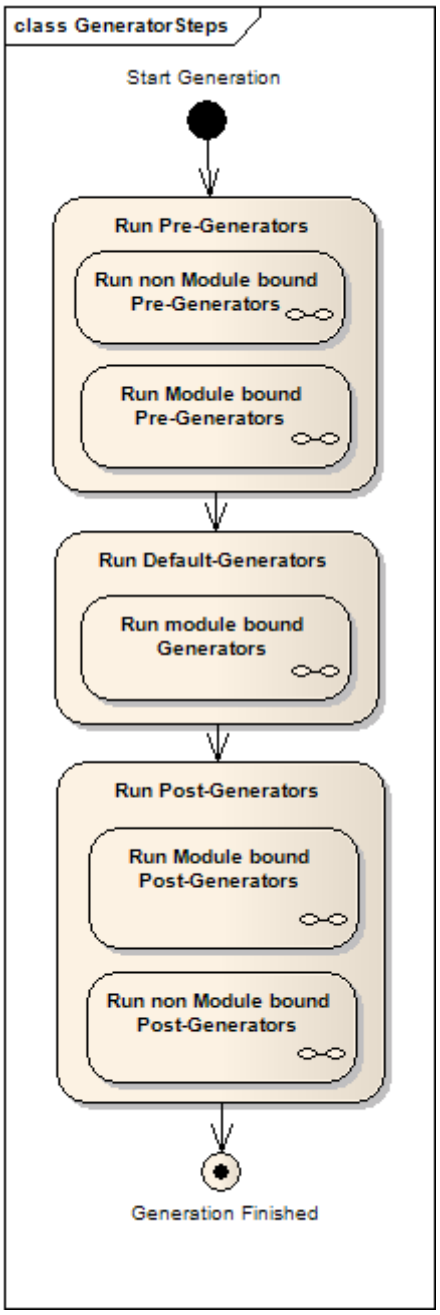


Figure 5.74. Generation Flow

5.17.3. JET Support

The NG Generator API includes support for generators based on JET (Java Emitter Templates) templates. The JET language is a scripting language similar to jsp that uses scriptlets with Java code to generate dynamic output between fix text. Before a JET template can be executed it must be processed by the JET builder that

generates a Java source file. This Java source file must first be compiled to a Java class before it can be executed.

The output of a JET generator normally results in one file. There are no restrictions on the Java expressions within a scriptlet so a JET generator can contain more powerful logic than a classic template based code generator could ever have. The following code snippet shows a simple JET file:

```
<%@ jet package="xyz.jet.generated"
        class="TheNumbersGame" %>
<%
  for( int i = 1; i <= 10; i++ ) {
%>
This is round <%=Integer.toString( i )%>.

<%
  }
%>
```

5.17.4. Standalone JET compiler

As said above a JET template must be generated to a Java source file first before using it. For that reason EB tresos Studio provides a standalone JET compiler application. You can run it via the `jet_compile` command located in the `bin` directory of your EB tresos Studio installation.

You can specify several space-separated JET template files as arguments to that command. The template files can either be specified absolute or relative to the current execution directory (the directory from where you start the application, normally the `bin` directory).

With the optional parameter `-o` it is possible to define the output directory for the resulting Java files. If the output directory is not defined then the files are generated to the current execution directory. Note that the resulting Java file is described by the `jet` directive within each JET template. The application takes respect to the package requirements of the Java class and creates an appropriate directory structure under the output directory.

```
jet_compile -o C:\work\generated ..\src\templates\MyJetFile.jet
```

5.17.5. C Data Structures Generator (CDS)

The C Data Structures (CDS) Generator is a completely new code generation framework. It provides functionality to generate type and variable definitions from Java objects into the C language. The CDS generator is

basically intended to generate configuration values into C-structs that are required by the runtime code of a module. By separating the configuration values from the logic it is possible to replace the configuration values after compiling the runtime code. So this feature targets to be used for link-time and post-build configurations.

The following features are supported:

- Primitive types, pointers, structs and arrays
- Typedefs
- Variable initializations of built-in and self-defined types
- Null pointer and function pointer support
- Compiler abstraction
- Memory abstraction
- Comments for each variable and member
- Volatile modifier
- Conditional members within structs

The CDS Generator contains Java classes that represent primitive types, structs, pointers and arrays of the C language. These classes can either be used directly to generate primitive variables or can be subclassed to create own types defined by a `typedef`.

The output is mainly controlled by so called backends. Currently the only backend provided by EB tresos Studio is the `CBackend` that writes pure C code. It is possible to extend the generator by providing various backend implementations, e.g. a binary backend.

5.17.6. Apache Ant Support

One of the main goals of the NG Generator API is the integration of the popular build tool Apache Ant. In fact, the NG Generator API always requires an Ant build file to execute. As mentioned above the NG Generator API uses the existing generator extension point. So the registered generator class must always be `dreisoft.tresos.generator.ng.api.NGGenerator`. The build file(s) are specified with nested `parameter` tags to the NGGenerator class at the generator extension point.

```
<extension point="dreisoft.tresos.generator.api.plugin.generator"
           id="NGGeneratorAPITest_TS_genreator_id"
           name="NGGeneratorAPITest_TS Generator">

  <generator id="NGGeneratorAPITestGenerator"
             moduleId="NGGeneratorAPITest_TS"
             class="dreisoft.tresos.generator.ng.api.NGGenerator">
```

```
    <parameter name="buildfile"
               value="resources/generator_build.xml" />
</extension>
```

There are Ant tasks provided for each type of generator. Together with the standard Ant tasks it is possible to create powerful build scripts. The following list gives a short overview of the delivered tasks:

Task	Description

`ng.property`	Allows to set an Ant property via a query to the DataModel:
	
	`<ng.property`
	`  name="propertyName"`
	`  dctxt="variableName"`
	`  xpath="XPath expression"`
	`  override="false" >`
	`</ng.property>`

`ng.setgeneratorvar`	Allows to set context variables from within the build.xml:
	
	`<ng.setgeneratorvar`
	`  name="propertyName"`
	`  dctxt="variableName"`
	`  xpath="XPath expression"`
	`  override="true" >`
	`</ng.setgeneratorvar>`

`ng.getgeneratorvar`	Allows to get context variables and set it to ant properties:
	
	`<ng.getgeneratorvar`
	`  name="variableName"`
	`  property="propertyName"`
	`  override="true" >`
	`</ng.getgeneratorvar>`

Task	Description

`ng.tmplgen`	Runs the template based code generator:
	
	`<ng.tmplgen`
	`  template="templates/CanIf_Cfg.h"`
	`  file="CanIf_Cfg.h"`
	`  dctxt="variableName"`
	`  failonerror="false"`
	`  needCodeTags="true" >`
	
	`  <include path="generate" />`
	
	`  <variable name="var1" value="foo" />`
	`  <variable name="var2" value="bar" />`
	`</ng.tmplgen>`

`ng.extgen`	Runs an external code generator:
	
	`<ng.extgen`
	`  commandline="$${pluginPath}/bin/gen.exe`
	`  -f $${outputDir}/epcFiles/project.epc`
	`  -o $${outputDir}/src/CanIf_Cfg.c"`
	`  cfgFilePath="epcFiles"`
	`  contentType="asc:2.0"`
	`  generateAllModules="true"`
	`  generateIntoOneFile="true"`
	`  workingDir="/TEMP"`
	`  shell="false"`
	`  timeout="5"`
	`  error="^\s*Error\[$$f,$$l\]:\s*$$m\s*$$"`
	`  warning="^\s*Warning\[$$f,$$l\]:\s*$$m\s*$$"`
	`  info="^\s*Info\[$$f,$$l\]:\s*$$m\s*$$"`
	`  failonerror="true" >`
	
	`  <variable name="var1" value="foo" />`
	`  <variable name="var2" value="bar" />`
	`</ng.extgen>`

Task	Description

`ng.jetgen`	Runs a jet template that is already compiled to a Java class that implements interface `dreisoft.tresos.generator.ng.api.ant.IJavaGenTemplate`:
	
	`<ng.jetgen`
	`  class="some.package.SomeJetGenerator"`
	`  file="JetOutput.txt"`
	`  failonerror="false" >`
	
	`  <argument name="useDet" value="true" />`
	`  <argument name="someArg" value="123" />`
	`</ng.jetgen>`

`ng.jetgen.src`	Compiles and runs a jet template that is delivered as source code:
	
	`<ng.jetgen.src`
	`  class="some.package.SomeJetGenerator"`
	`  file="JetOutput.txt"`
	`  failonerror="true" >`
	
	`  <jetIncludes`
	`  dir="../templates"`
	`  includes="**/*Generator.jet" />`
	`  <javaIncludes dir="../src" includes="**/*.java" />`
	
	`  <argument name="someArg" value="123" />`
	`  <argument name="arg1" value="${someAntProperty}" />`
	`</ng.jetgen.src>`

`ng.javagen`	Runs a java generator. It must implement `dreisoft.tresos.generator.ng.api.ant.IJavaGenTemplate`:
	
	`<ng.javagen`
	`  class="some.package.JavaGenGenerator"`
	`  file="generated.txt" >`
	`</ng.javagen>`

Task	Description

`ng.javagen.src`	Compiles and runs a java generator that is delivered as source code:
	
	`<ng.javagen.src`
	`  class="some.package.JavaGenGenerator"`
	`  file="xyz.txt"`
	`  failonerror="false" >`
	
	`  <includes dir="../src" includes="**/*.java" />`
	
	`  <argument name="someArg" value="abc_" />`
	`  <argument name="arg1" value="${someAntProperty}" />`
	`</ng.javagen.src>`

Table 5.49. NG Generator Ant tasks

5.17.7. Source Delivery

As shown in the Ant tasks overview above there are the tasks `ng.javagen.src` and `ng.jetgen.src` that allow to deliver the generator as source code and not already compiled to a class. In case of the Java-based generator it is the Java source file that implements the `dreisoft.tresos.generator.ng.api.ant.IJavaGenTemplate` interface, in case of the JET-based code generator it is the JET template.

Both tasks first compile the specified source files before executing the specified generator class. The required Java compiler is delivered with the NG Generator API and it is not possible to use another one. So the generator source code must be compatible with the relevant compiler (current version Sun SDK 1.5.0). The class path is based on the plugin's class path. It is possible to add further class path locations. The source files are compiled to a temporary directory within the configuration area of EB tresos Studio.

5.18. Setting Up Eclipse

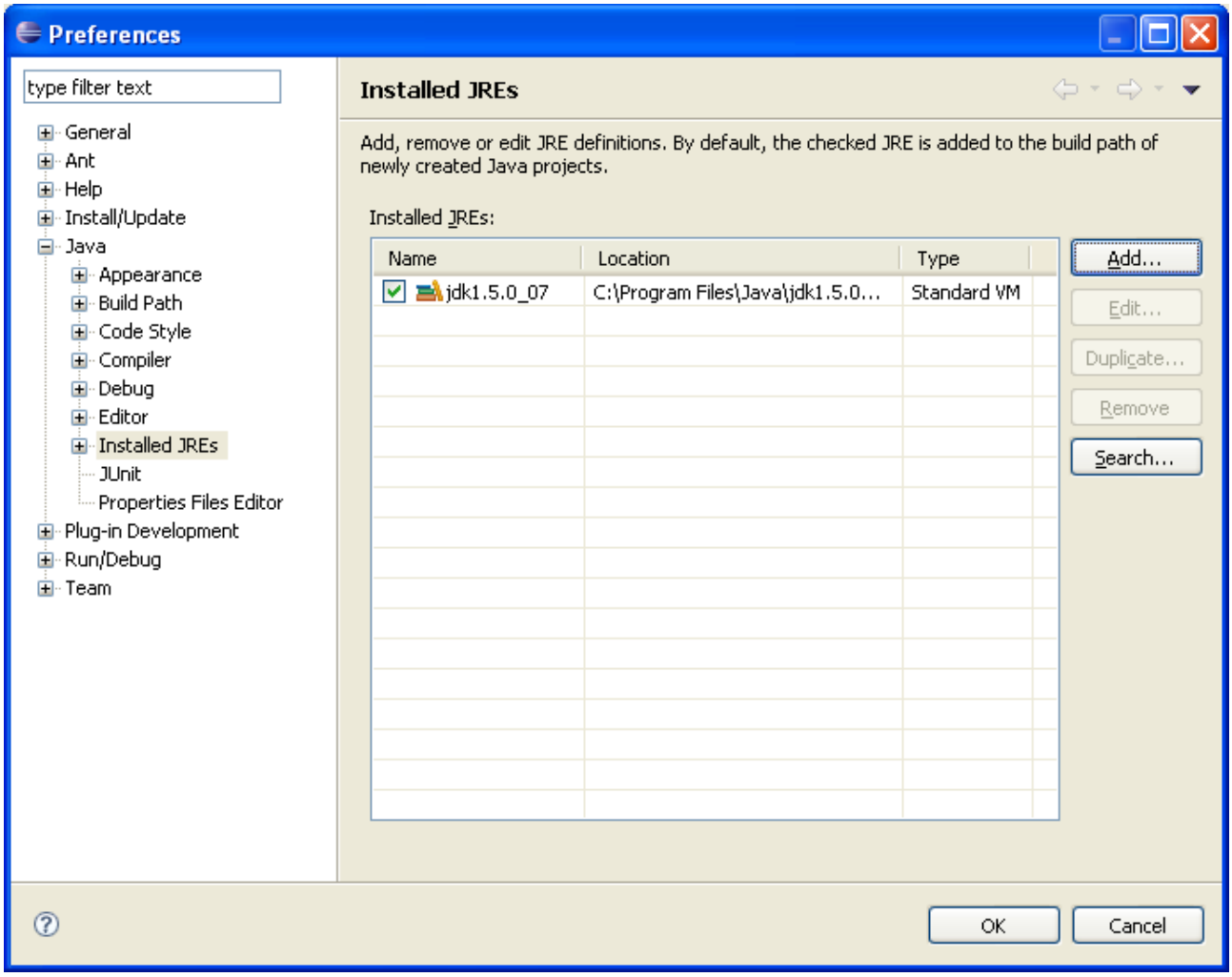
5.18.1. Installation And Version

EB tresos Studio is built on *Eclipse 3.3.1* and *Java JRE 5.0 Update 16*. Thus it is best to also use these versions for the development of EB tresos Studio. A copy of Eclipse can be downloaded from the Eclipse Download

Page. A Java 5.0 Update 16 can be downloaded from the Java Archive Site. Download the JDK instead of the JRE, so you also have the Java sources for debugging purposes.

To install Eclipse just unzip the downloaded zip file to your harddrive.

Now the Java Environment has to be setup. Start Eclipse and go to *Window -> Preferences -> Java -> Installes JREs* and add the top-level-directory of your JDK installation.



5.18.2. Installation of EB tresos Studio Eclipse Tooling

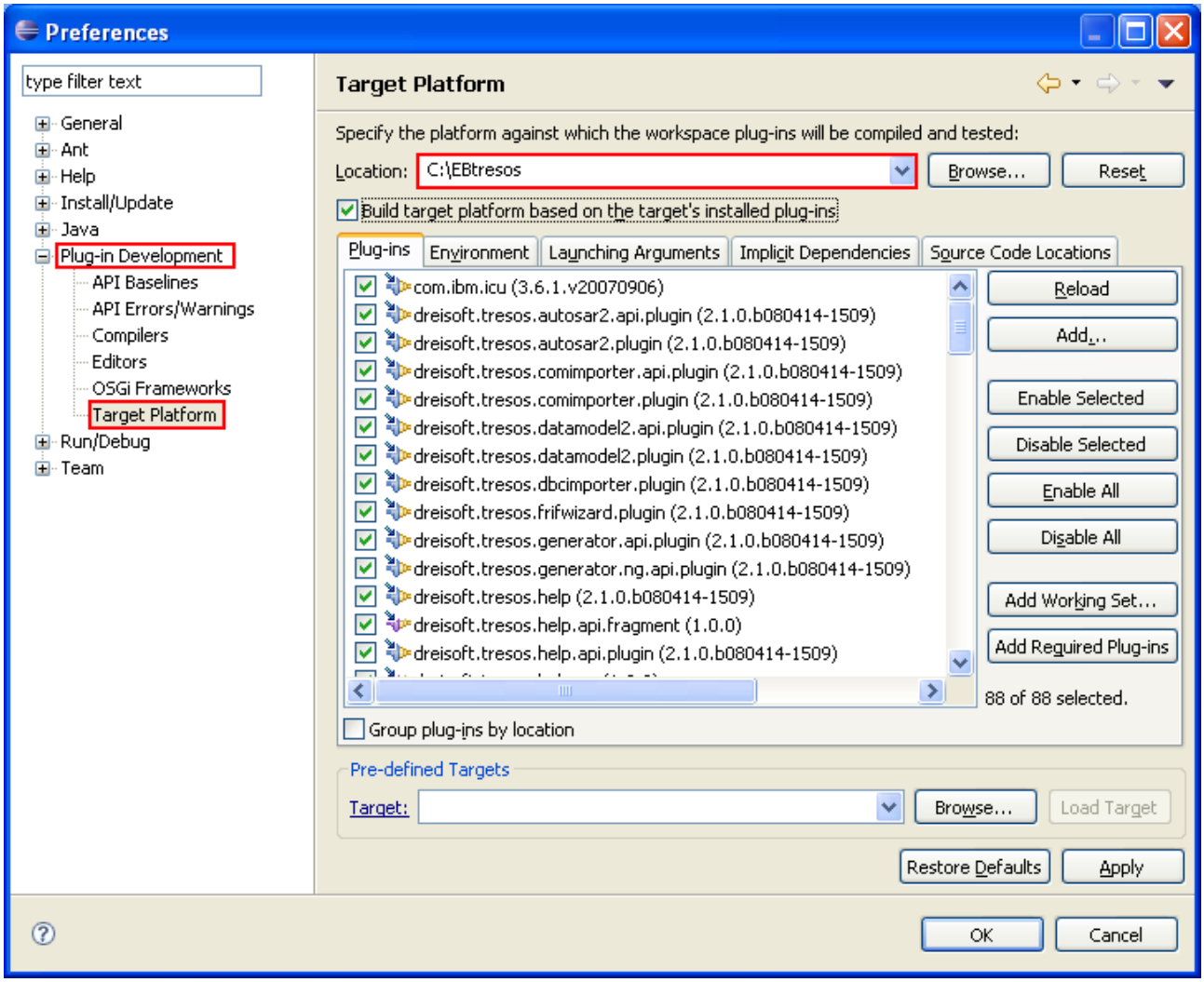
To use some special utilities, that are necessary for the programming with the EB tresos Studio Public API, go to the EB tresos Studio installation directory and unzip the file *tools/eclipse_tools.zip* to your eclipse installation. The zip-file contains 3 plugins and a .exe file.

- dreisoft.tresos.help.api.jar - which integrates the EB tresos Studio Public API help into Eclipse's help system.

- dreisoft.tresos.sdk.api.jar - EB tresos Studio Software Development Kit that generates NLS/OS for a project (Section 5.19: National Language Support (NLS) / Operation Status (OS)).
- dreisoft.tresos.lib2.jar - Library needed by the SDK.
- generate.exe - adds the ability to generate NLS/OS from the commandline

5.18.3. Setting Up The Target Platform

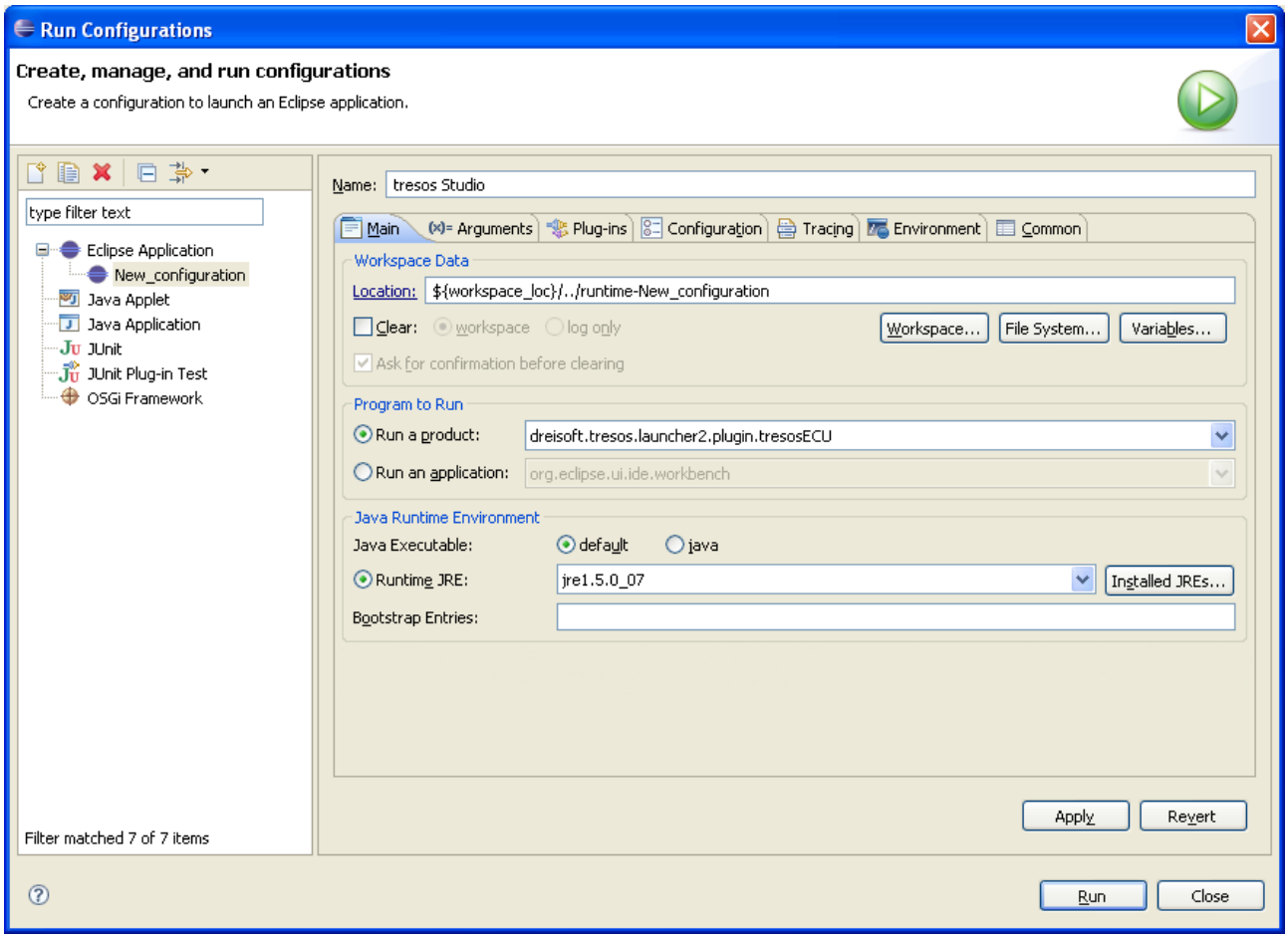
To be able to start EB tresos Studio from within Eclipse you need to setup the target platform. This can be done by going to *Window -> Preferences -> Plug-In Development -> Target Platform*. Set the *Location* to where your EB tresos Studio installation is located and press OK. Done.



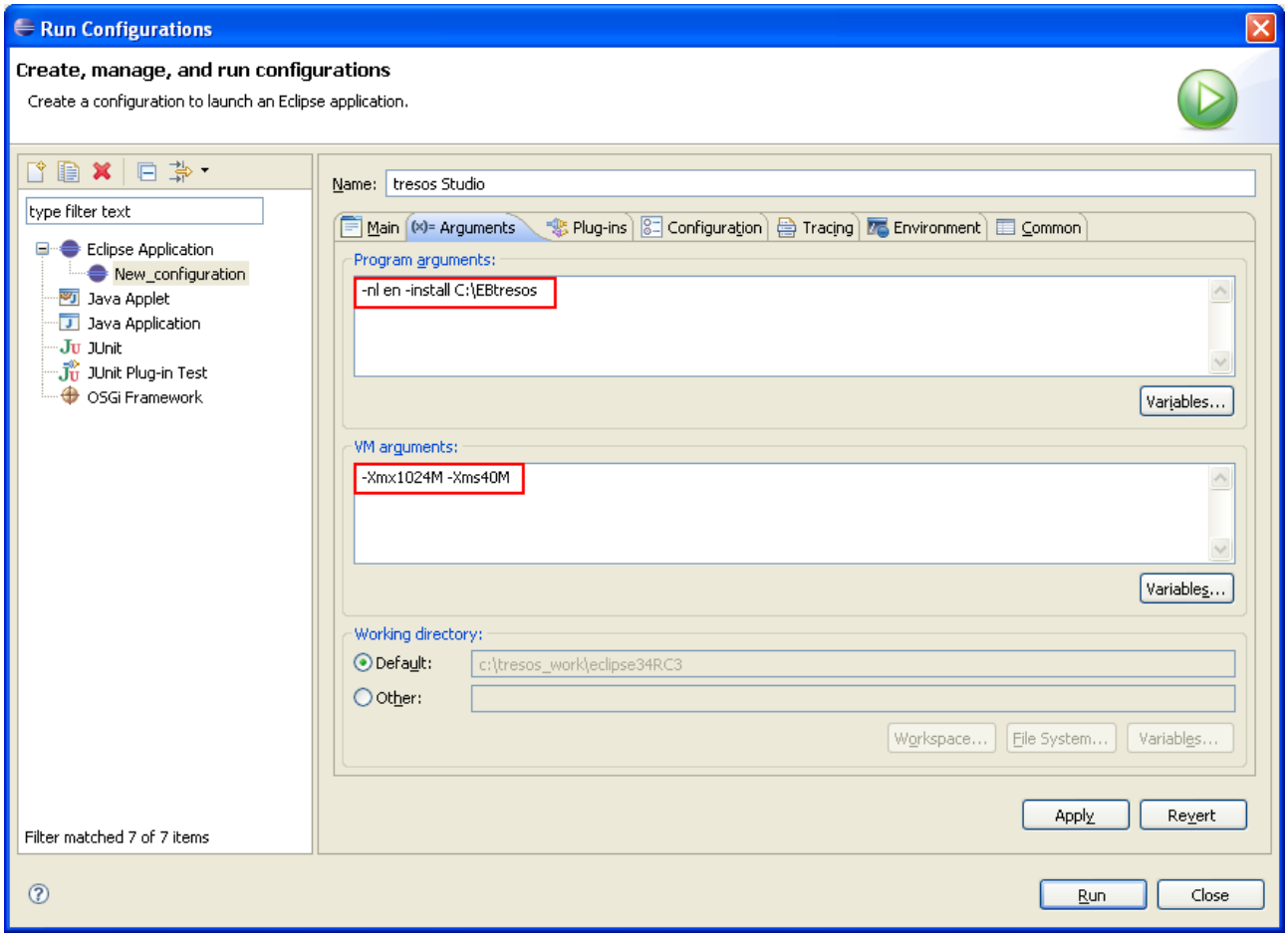
5.18.4. Setting Up A Run Configuration

The last step for setting up Eclipse to start tresos is to create a *Run Configuration*. To do so go to *Run -> Open Run Dialog*. Double Click on *Eclipse Application* and give it a name.

The following images show what has to be done to setup a correct *Run Configuration*.



Choose "dreisoft.tresos.launcher2.plugin.tresosECU" from the product-combobox as shown in the picture above.

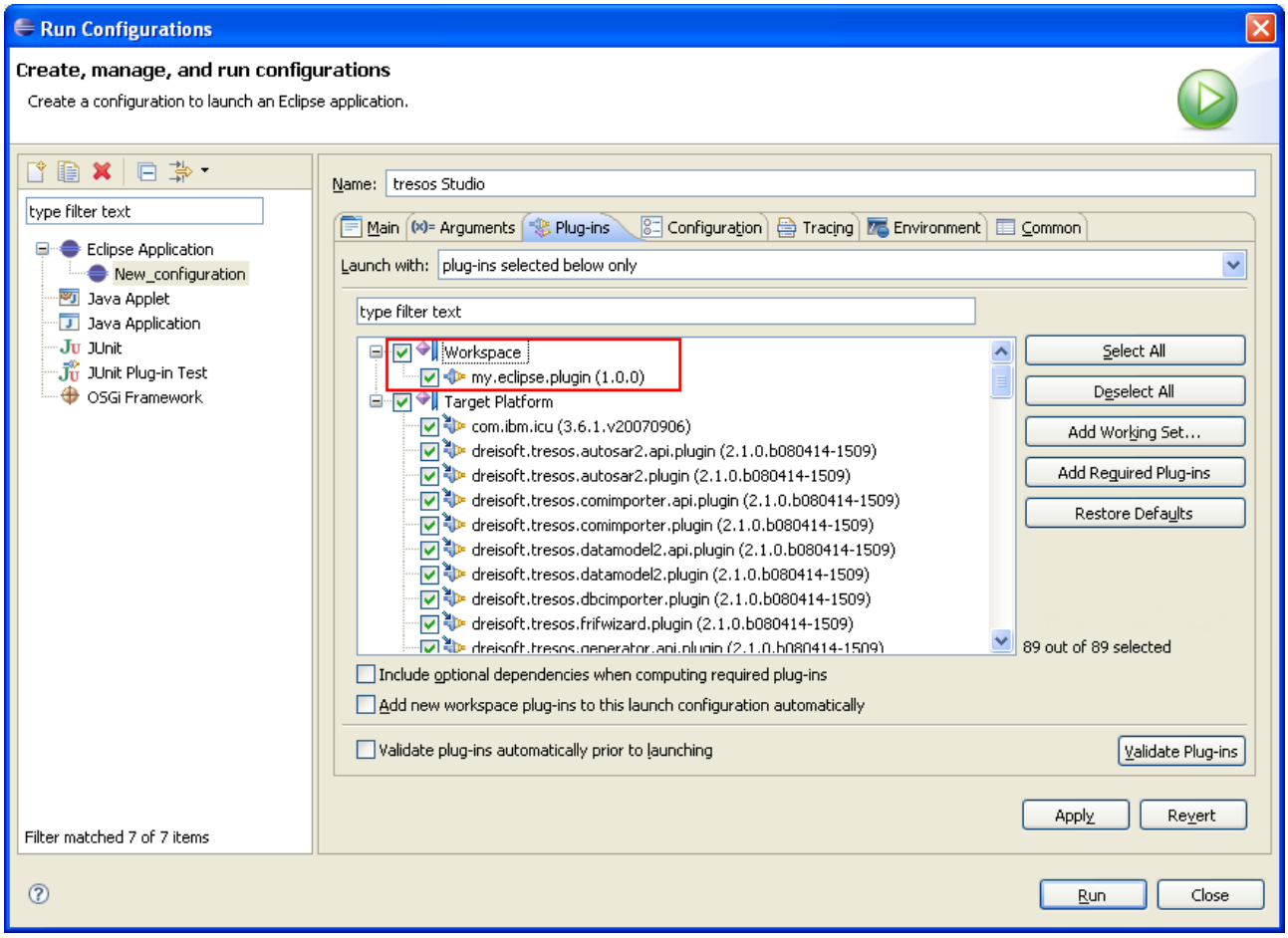


On the Arguments-tab, add "-nl en -install <tresos-basedir>" to the Program arguments. The given tresosbasedir is only used to get the license from the contained bin-directory. Add "-Xmx1024M -Xms40M" to the "VM arguments" section.

Tip

Caching-problems

Because eclipse is caching some things, it might be possible, that you cannot see changes made within your ModulePlugIn in the next run of EB tresos Studio. To solve this problem, enable the checkbox "Clear workspace data" within the run-dialog.



Set *Launch with* to *plug-ins selected below only*. Select *all* Target Platform plug-ins. From your Workspace just select that plug-in you want to run with EB tresos Studio. In this case it is the demo plug-in shipped with EB tresos Studio (demos/PublicApiTest_TS_T00D0M0I0R0).

5.19. National Language Support (NLS) / Operation Status (OS)

5.19.1. National Language Support (NLS)

National Language Support is used to localize messages or labels of a plug-in.

For eclipse plugins, there are two places where localized messages are most likely used: in the plugin manifest file and inside the java code. In the java code, NLS can easily be used by calling a method that returns the

localized version of a string. Parameterized messages are easily achievable through method parameters. The EB tresos Studio NLS framework therefore generates classes that provide such methods.

However, in the plugin manifest files (`plugin.xml` and `Manifest.mf`) no generated java methods for translation can be used. EB tresos Studio therefore adopts the eclipse approach, which is explained in the next paragraph, and extends it a bit.

The eclipse approach works as follows: inside the plugin manifest properties can be used which are automatically replaced by localized strings at runtime. The properties have the following form: `%propertyName`. The values of the properties are looked up at runtime from *.properties files. If a locale is set, the lookup takes place in the file `plugin_<locale>.properties`. If no such file exists, or if a single message cannot be found inside that file, default file called `plugin.properties` is used as fallback. The syntax of the property files follows the usual eclipse property file format.

5.19.1.1. Workflow for NLS-support in Java Code

The EB tresos Studio NLS framework uses classes extending the `dreisoft.tresos.lib2.nls.NLS` class for the retrieval of localized messages. Those classes are automatically generated from textfiles with the `NLS/OS-Tool` program in the `dreisoft.tresos.sdk.api` plugin.

Using the NLS framework in Java code roughly involves the following steps:

- define a `nls-<classname>.txt` config file (with `<classname>` being the name of the generated class in the following step). A description of the config file format and an example of how a config file should look like, can be found in Section 5.19.1.2: Config File Format
- after creating a config file, open the properties of your plugin (right-click -> Properties) and go to the NLS Preferences tab. There you need to select the Java source directory and then "Add" the config file to the table below. If the Type, shown in the table, is not set to "nls" do that manually.

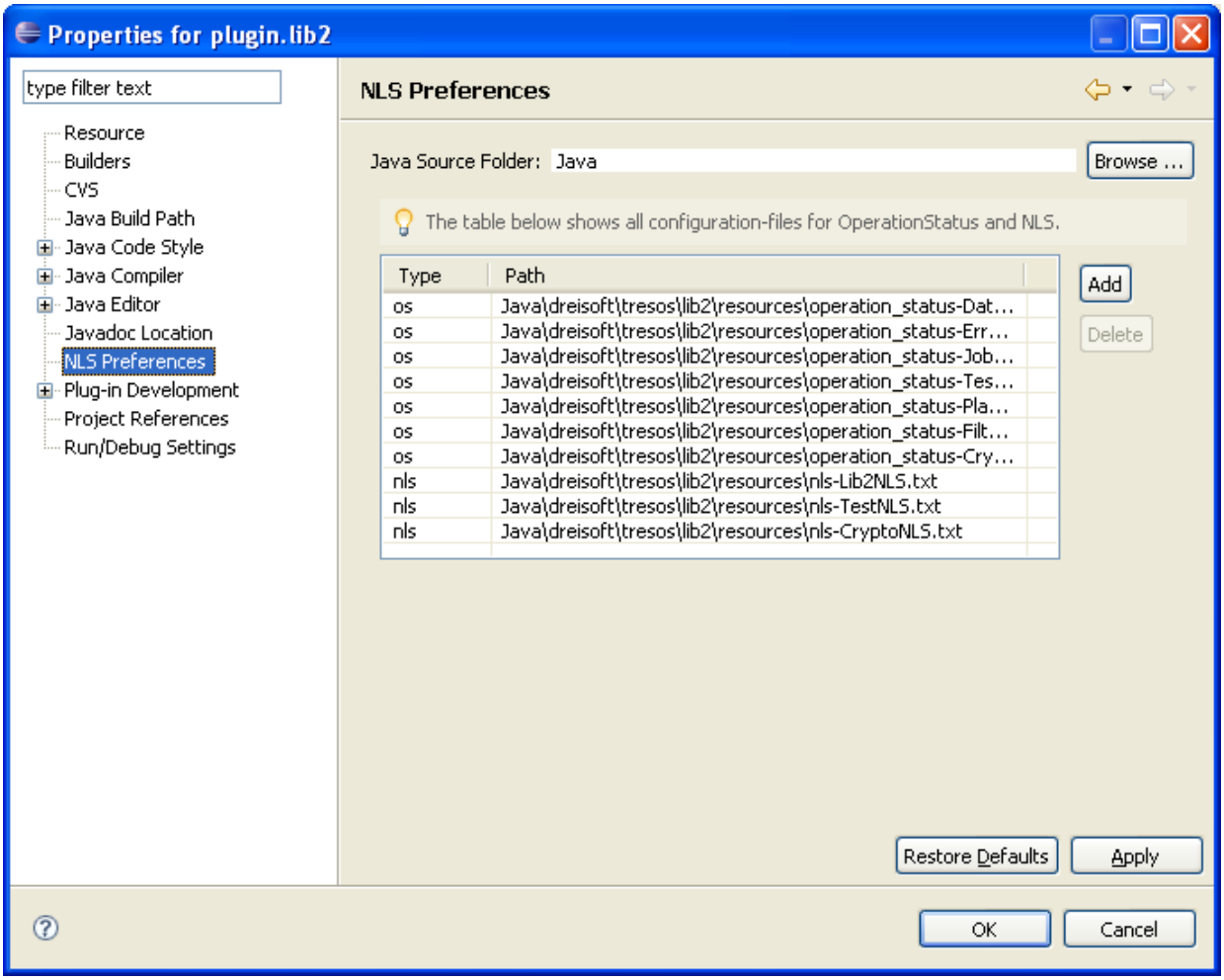


Figure 5.75. NLS/OS-Tool Preference Page

- use the NLS/OS-Tool to auto-generate the property files and classes. See section Section 5.19.3: Calling The NLS/OS-Tool for information on how the tool can be called.
- use the static methods of the generated class `<classname>` in your code to generate the localized strings

5.19.1.2. Config File Format

The config file consists of two parts: a header section which provides general info necessary for class generation, followed by a list of message blocks wich begin with the message id and contain the messages and additional data. Each line inside the config file begins with a directive. If a `#` character is found a the beginning of a line, the line is regarded as comment and ignored. Whitespace is filtered out automatically.

In the following the syntax of a config file is explained.

```
# The NLS factory class to be generated
TARGET-CLASS: test.tresos.plugin.generated.nls.MyNLSClass
# Note, that this class must be derived from APINLS
EXTENDS     : APINLS
# Note, that this class must import dreisoft.tresos.lib2.api.log.APINLS
IMPORT      : dreisoft.tresos.lib2.api.log.APINLS
# Optional: if the generated class should implement one or several
# Interfaces, they can be added here separated by a semicolon
IMPLEMENTS: ISomeClass
# The imports are needed for the different parameters
# of the factory methods (see PARAMs below)
IMPORT : dreisoft.tresos.datamodel2.api.DCtxt
# Optional: The javadoc-description of the generated class can be edited
# here
DESCRIPTION: This class contains several status for application-specific
errors and warnings
# specifies the registration-name under which the resource-bundle
# is registered at the ResourceHandler and the full-qualified
# properties-file-name (to which all messages will be written)
# <Bundle-Name>=<File-Name>
BUNDLE-NAME: MyEclipsePluginNLS=test.tresos.plugin.resources.
             MyEclipsePluginNLS
# Additional code can be added (only) at the end of the file
# using the following tag. All lines up to the end of this
# tag will be added as code to the generated class

Example:
TARGET-CLASS: test.tresos.plugin.generated.nls.MyNLSClass
BUNDLE-NAME : MyEclipsePluginNLS=test.tresos.plugin.resources.
              MyEclipsePluginNLS
IMPORT      : dreisoft.tresos.lib2.api.log.APINLS
EXTENDS     : APINLS
```

The header can be followed by multiple message blocks. Each block contains the unique message id, the nonlocalized message, the parameter definitions of the message and possibly multiple translations of the message.

```
ID      : HELLO_WORLD
PARAM   : String arg0=an adjective
PARAM   : String arg1=another adjective
L-en    : Hello {0} {1} England!
L-de    : Hallo {0} {1} Deutschland!
```

ID

The ID is used as the name of the factory method and also used to generate a constant variable with the name `ID_<Id>` that stores the id of the message. The ID must be unique within one file.

PARAM

A PARAM defines a parameter with which the message can be parameterized. The parameter has the following form:

```
"PARAM" ":" [ [ <Class> ] <parameterName> "=" ]
<description>
```

The Class must be added as IMPORT within the file-header. Primitive types (e.g. int, float) can be used too. The default-Case is String. If no parameterName is given, a name of the form `<param><number>` will be generated.

Example:

```
PARAM : INode node=the node to describe
PARAM : value=a string-value
PARAM : a description for a string-parameter
```

L-*

The L-* are the localized messages where * is the name of the language (e.g. "en" for English, which is mandatory). The value can contain parameter references to the parameters defined with the PARAM keyword in the form defined by the `java.text.MessageFormat`.

5.19.1.3. NLS Examples

5.19.1.3.1. Example for the usage in a Plugin.xml

In a plugin.xml no Java methods can be called to localize your plugin. So some special settigs must be made. At first the plugin must be told where to find the localization file. This is done within the `MANIFEST.MF` with the following entry:

```
  Bundle-Localization: plugin
```

Then a file named nls-plugin.txt must be created somewhere in your plugin with similar entries like the following:

```
#
# NLS Messages for the plugin.xml file
#

MODE: MODE_PROPERTIES

ID  : DescriptionPage
L-en: englisch description
L-de: deutsche Beschreibung
```

The nls-plugin.txt gets a special handling from our NLS/OS class generator (Section 5.19.3: Calling The NLS/OS-Tool) and only generates *.properties files in the root dir of the plugin.

At last the NLS messages can be used in the plugin.xml. Eclipse gets the NLS information from the generated .properties files and replaces the %NLS.placeholder with the localized values.

```
<extension
        id="DescriptionPage"
        name="DescriptionPage"
        point="org.eclipse.ui.views">

  <view
```

```
          id="package.name.of.this.DescriptionPage"
          name="%NLS.DescriptionPage"
          icon="icons/description.gif"
          class="package.name.of.this.DescriptionPage"
          category="yourapp"
          allowMultiple="false"/>
</extension>
```

5.19.1.3.2. NLS source code usage example

To use NLS in source code, no special entry is needed. Take the following example of the creation of a button, where you want to set a localized text. At first you need the NLS config file nls-MyEclipseNLS.txt from which the Java class is generated by the NLS/OS tool class generator (Section 5.19.3: Calling The NLS/OS-Tool).

```
TARGET-CLASS: test.tresos.plugin.generated.nls.MyEclipseNLS
BUNDLE-NAME : MyEclipse=test.tresos.plugin.generated.resources.MyEclipse
IMPORT      : dreisoft.tresos.lib2.api.log.APINLS
EXTENDS     : APINLS

#########################
#
# Section Comment
#
#########################

ID      : BUTTON_BROWSE
L-en    : Browse
L-de    : Durchsuchen
```

```
m_button = new Button( parent, SWT.PUSH );
m_button.addSelectionListener( new SelectionListener() {
    public void widgetSelected( SelectionEvent e ) {
        // do something
    }
```

```
    public void widgetDefaultSelected( SelectionEvent e ) {
    }
} );
// localized message
m_button.setText( MyEclipseNLS.BUTTON_BROWSE() );
// non-localized message
// m_button.setText( MyEclipseNLS.NONLS_BUTTON_BROWSE() );
```

5.19.2. APIOperationStatus

There are a lot of complex and nested operations possible within the DataModel. Each of these operations can return an error, a warning or an info-message, so that the result of an operation may be a complex structure of informations.

Depending on the situation, the system must be able to provide the right informations at the right time. For example, if the user starts the generation and there is an error within the loaded files, only one error-message-line is needed including the information, where the error occured. On the other side - if the user cannot solve this problem - there must be more informations available for the programmer, e.g. the call stack of the occured exception within a log file. Maybe informations about previously successful operations are needed too. To handle these requirements, the Lib2.API provides a sophisticated error reporting system.

In EB tresos Studio all errorhandling is done with APIOperationStati. Stati are handed to methods, and methods just hang in their own status if an error occured. The calling method then checks the status after the called method returned. An example on how to use an APIOperationStatu can be found in section Section 5.19.2.5: Operation Status Examples.

5.19.2.1. Glossary

Term	Description
APIOperationStatus	All status information is gathered in an operation status. An operation status is a container that stores a tree of status messages.
Status Message	One entry in an operation status consisting of all data of the message like, id, parameters, severity.
Status Code	A status message is uniquely identified by a status code which is a 32 Bit Integer or a String, depending on the PREFIX in the Configuration File.
Configuration File	All messages available in the system are defined in configuration files which are simple ASCII-files defining all static data of a status message.

Table 5.50. Glossary Statushandling

5.19.2.2. Functionality

- There are four severity-levels for messages:

  Error:
  A real problem occurred. The operation cannot be completed successfully.

  Warning:
  A possible problem occurred. The operation can be completed, but the result may have problems.

  Info:
  An informative message.

  Ok:
  This is not a real status but is used to describe the running operation to provide a context for nested messages.
- Because operations may be nested, the messages may reflect this structure. E.g. if there are 5 nested operations and the last operation generates an error-message, the message must includes context informatio from all 5 nesting levels.
- A message may provide a localized version for the user (e.g. in japanese language, french etc.).
- A message provides a non-localized representation for log files.
- An error-message includes a stack trace for debugging purposes.
- As a status is created very often the status handling is time performance optimized.
- Messages are configured with id, severity, non localized version, localized versions, parameter specification via a textfile.
- Messages may be parameterized to provide context information provided by the creator of the message.

- Messages may contain a node reference so an error can be navigated to in a GUI based application.
- The API provides a typesafe interface for reporting messages by using a code-generator to generate message classes.

5.19.2.3. Workflow for OS-support in Java Code

The EB tresos Studio OS framework uses classes extending the `dreisoft.tresos.lib2.api.log.API-OperationStatus` class for creation of Operation Stati. Those classes are automatically generated from textfiles with the `NLS/OS-Tool` program in `dreisoft.tresos.sdk.api` plugin.

Using the Operation Status framework in Java code roughly involves the following steps:

- define a `operation_status-<classname>.txt` config file (with `<classname>` being the name of the generated class in the following step). A description of the config file format and an example of how a config file should look like, can be found in Section 5.19.2.4: Config File Format
- after creating a config file, open the properties of your plugin (right-click -> Properties) and go to the NLS Preferences tab. There you need to select the Java source directory and then "Add" the config file to the table below. If the Type, shown in the table, is not set to "os" do that manually.

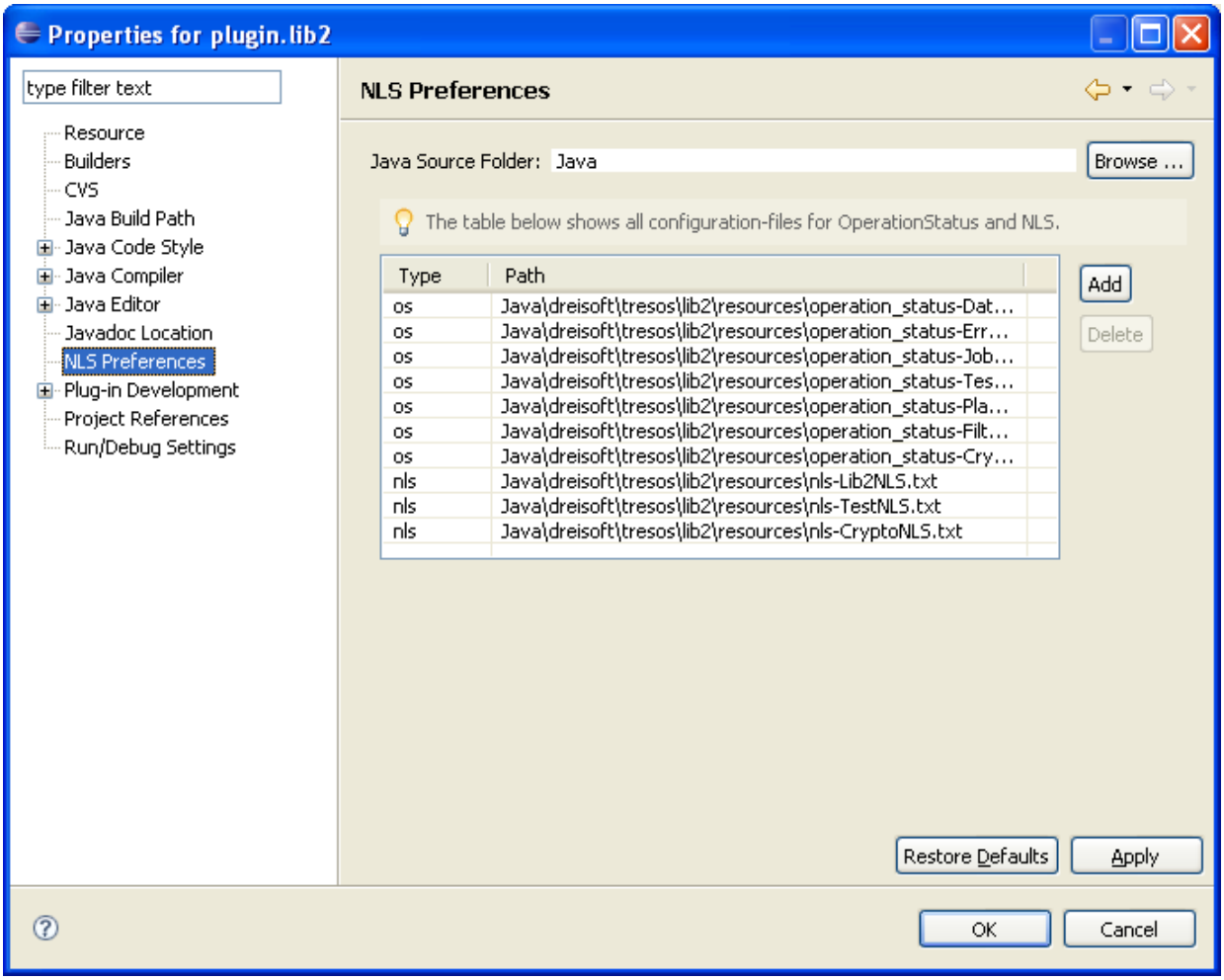


Figure 5.76. NLS/OS-Tool Preference Page

- use the NLS/OS-Tool to auto-generate the property files and classes. See section Section 5.19.3: Calling The NLS/OS-Tool for information on how the tool can be called.
- use the static methods of the generated class `<classname>` in your code to generate the localized strings

5.19.2.4. Config File Format

Each configuration file will be used to create one java-class derived from the OperationStatus-class.

Configuration files are written in the syntax of Java property files which contain tag/value pairs with a ":" as delimiter. A # at the line beginning can be used for comments. Empty lines are ignored. A configuration file starts with a header followed by status message definitions.

The header of a configuration file must contain the following configuration-parameters:

```
# this integer will be added to all codes - the combination
# PREFIX+CODE must be unique for all operation stati in the system
# This PREFIX can be an Integer or a String. In case it is an Integer
# the error Code of an Operation Status will be added to the prefix.
# In case it is a String, the ERROR Code will be appended to the prefix
# with an underscore
PREFIX : 1000
# PREFIX : ASC

# The operation status factory class to be generated
TARGET-CLASS: test.tresos.plugin.generated.operationstatus.MyOperationStatus

# Optional: if the generated class should extend another class
# as OperationStatus, this class can be given here
# Note, that this class must be derived from OperationStatus
EXTENDS: MyAbstractOperationStatus

# Optional: if the generated class should implement one or several
# Interfaces, they can be added here separated by a semicolon
IMPLEMENTS  : ISomeClass

# The imports are needed for the different node types and
# parameters of the factory methods (see PARAMs below)
IMPORT : dreisoft.tresos.datamodel2.api.DCtxt

# Optional: The javadoc-description of the generated class can be edited here
DESCRIPTION : This class contains several status for
application-specific errors and warnings

# specifies the registration-name under which the resource-bundle
# is registered at the ResourceHandler and the full-qualified
# properties-file-name (to which all messages will be written)
# <Bundle-Name>=<File-Name>
BUNDLE-NAME : status=test.tresos.plugin.resources.OperationStatus

  Example:
  PREFIX      : 1000
  TARGET-CLASS: dreisoft.generated.operationstatus.MyOperationStatus
  BUNDLE-NAME : status=dreisoft.generated.resources.OperationStatus
```

The header is followed by the definition of the status messages. A complete entry for one message looks like this:

```
ID : EXAMPLE_STATUS
CODE : 1
SEVERITY: ERROR
PARAM : String value=description of parameter 0
PARAM : DCtxt ctxt=description of parameter 1
L-en : the english message with two parameters: {0} {1}
L-de : die deutsche Meldung mit zwei Parametern: {0} {1}
```

ID

The ID is used as the name of the factory method and also used to generate a constant variable with the name `CODE_<Id>` that stores the code of the message. The ID must be unique within one file.

CODE

The CODE is the status code of the status message. The code is a 32bit integer. The prefix defined in the header is added to the CODE to form the status code. The status code must be unique in the system.

SEVERITY

The severity level of the messag: ERROR, USER_ERROR, WARNING, USER_WARNING, INFO. If the SEVERITY start with USER_ the message is flagged as error type user error or user warning.

PARAM

A PARAM defines a parameter with which the message of an operation status can be parameterized. The parameter has the following form:

```
"PARAM" ":" [ [ <Class> ] <parameterName> "=" ]
<description>
```

The Class must be added as IMPORT within the file-header. Primitive types (e.g. int, float) can be used too. The default-Cass is String. If no parameterName is given, a name name of the form `<param><number>` will be generated.

Example:

```
PARAM : INode node=the node to describe
PARAM : value=a string-value
PARAM : a description for a string-parameter
```

L-*

The L-* are the localized messages for the status message where * is the name of the language (e.g. "en" for English, which is mandatory). The value can contain parameter references to the parameters defined with the PARAM keyword in the form defined by the `java.text.MessageFormat`.

5.19.2.5. Operation Status Examples

The following example of an ExternalGenerator shows how an APIOperationStatus is used normally.

```
public void generate( CodeGeneratorContext context ) {

    int number = context.getDCtxtVariable().var.getInt( "path/to/a/number" );
    APIOperationStatus status = APIOperationStatus.getOkStatus();
    checkNumber( number, status );

    if(!status.isOk()) {
        // do something else
    }
}
public void checkNumber( int number, APIOperationStatus status ) {

    APIOperationStatus s = APIOperationStatus.getOkStatus();

    if( number > 100 ) {
        s = new MyEclipseOperationStatus.NUMBER_ERROR_STATUS( number );
    }
    // do something else

    // add the s to status at the end
    if(!s.isOk()) {
```

```
        status.addStatus(s);
    }
}
```

The corresponding APIOperationStatus config-file should look like the following.

```
# Copyright (C) EB Automotive - All rights reserved
#
# OperationStatus Messages for Autosar plugin handling.
# The structure of the file is described in the corresponding design-document
#

PREFIX        : EXAMPLE
TARGET-CLASS  : test.tresos.plugin.generated.operationstatus.
                  MyEclipseOperationStatus
BUNDLE-NAME   : MyEclipseOperationStatus=test.tresos.plugin.generated.
                  resources.APIOperationStatus
EXTENDS       : dreisoft.tresos.lib2.api.log.APIOperationStatus

#
# ConfigManager
#

ID      : NUMBER_ERROR_STATUS
CODE    : 1
SEV     : ERROR
PARAM   : int number=the number which is wrong
L-en    : A wrong number was detected: {0}
```

5.19.3. Calling The NLS/OS-Tool

As prerequisite the dreisoft.tresos.sdk.api plugin must be installed to the eclipse installation directory. The plugin can be found in the tresos installation in the directory "tools" as `eclipse_tools.zip`. Just unpack it to your eclipse installation dir.

call the NLS/OS-Tool via a button in the eclipse toolbar to auto-generate the NLS property files and classes(including `<classname>`)

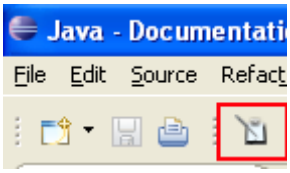


Figure 5.77. NLS/OS-Tool

The NLS/OS-Tool can also be called via the command line. For this use the generate.exe in the eclipse installation directory.

```
generate -p <path_to_the_plugin_project>
```

or directly via the eclipse.exe

```
eclipsec -nosplash -application dreisoft.tresos.sdk.api.plugin.NLSToolCommandline
<path_to_the_plugin_project>
```

This will call the NLS/OS-Tool for the project specified.

5.20. Public Api Use-Cases

A Demo Plugin is available under <tresos-install-loc>/demos/PublicApiTest_TS_T00D0M0I0R0 that can be installed as described here Section 5.22: Installing the Public Api Demo Plugin

5.20.1. General Preparations To Use The Public API

A plug-in which shall be fully Public API capable should have dependencies to the following EB tresos Studio plug-ins.

- dreisoft.tresos.autosar2.api.plugin
- dreisoft.tresos.datamodel2.api.plugin
- dreisoft.tresos.launcher2.api.plugin
- dreisoft.tresos.lib2.api.plugin
- dreisoft.tresos.generator.api.plugin

5.20.2. Using The Public API Generator

To register a Public API Generator for a module, two steps have to be done.

- Register the extension point *dreisoft.tresos.generator.api.plugin.generator*
- Implement the Public API Generator Java class.

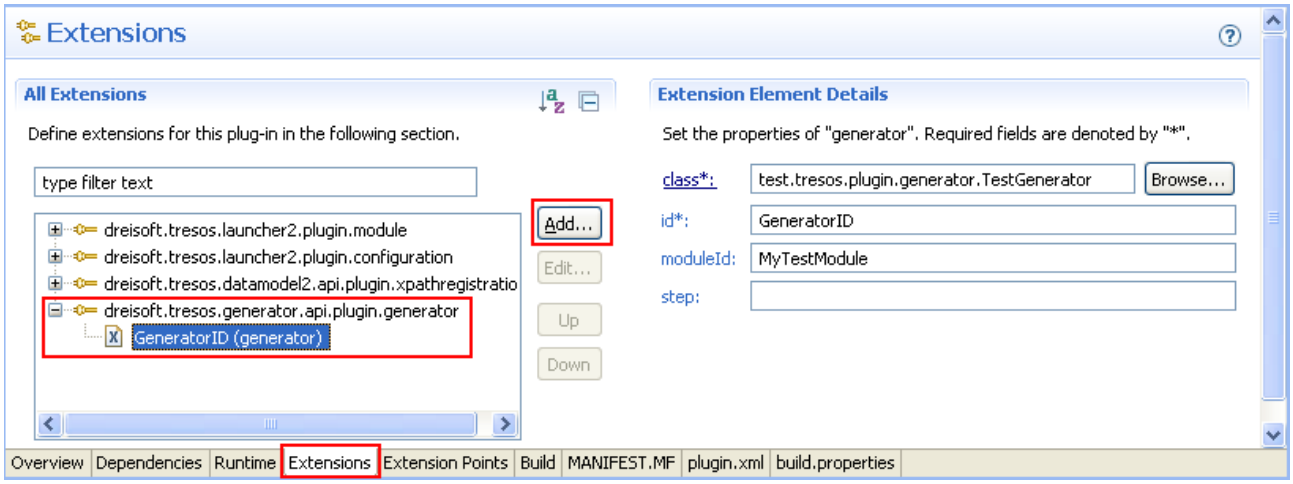
How this can be done will be explained in Section 5.20.2.1: Registering A Public API Generator.

5.20.2.1. Registering A Public API Generator

- Open the plugin.xml of your plug-in and switch to the *Extensions*-tab
- Click *Add* and select the *dreisoft.tresos.generator.api.plugin.generator* extension-point
- On the right-hand side enter a generator ID, the module ID to which this generator should be registered, as well as the step the generator should run in. Steps can be *pre*, *post* or *default* (leave blank for default).
- Create a class by clicking on *class*:* or *Browse...* to find an already implemented Generator. How to create a Generator will be explained in Section 5.20.2.2: Implementing A Public API Generator.

Tip

The moduleID can also be left blank. In this case it is a module independent generator that will be run on each generation run, for each module. But a module independent generator is only allowed for steps pre and post.



5.20.2.2. Implementing A Public API Generator

An external generator must implement the following Interface `dreisoft.tresos.generator.api.generator.ICodeGenerator` and the three methods:

- public void generate(CodeGeneratorContext context)

- public String getDefaultMode()
- public String[] getAvailableModes(ParameterSet parameters)

Examples:

```
/*
 * returns for which modes, the generator can be called
 */
public String[] getAvailableModes( ParameterSet parameters ) {
    return new String[]{MODE_GENERATE};
    // or new String[]{MODE_GENERATE, MODE_VERIFY} if you also want to verify
    // with this generator
}

/*
 * returns the default mode of the generator - In this case "generate"
 */
public String getDefaultMode() {
    return MODE_GENERATE;
}
```

An example for `public void generate( CodeGeneratorContext context )` can be found in the class TestGenerator in the test.tresos.plugin in the package `test.tresos.plugin.generator`.

For detailed information on the Public API Generator consult the Public API Java Documentation. If you installed the eclipse_tools.zip into your eclipse you can call it with *Help -> Help Contents -> Public API Java Documentation*

5.20.3. Using Custom XPath Functions

To register custom XPath functions, two steps have to be done.

- Register the extension point *dreisoft.tresos.datamodel2.api.plugin.xpathregistration*
- Implement the Java class that inherits the XPath functions.

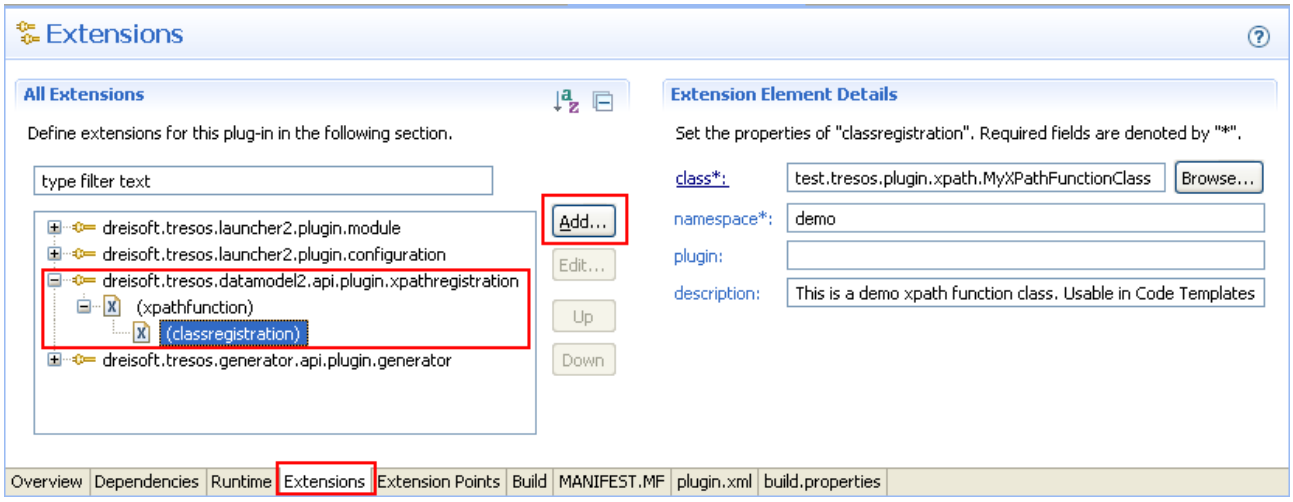
How this can be done will be explained in Section 5.20.3.1: Registering Custom XPath Functions.

5.20.3.1. Registering Custom XPath Functions

- Open the plugin.xml of your plug-in and switch to the *Extensions*-tab
- Click *Add* and select the *dreisoft.tresos.datamodel2.api.plugin.xpathregistration* extension-point
- On the right-hand side enter a namespace under which the xpath functions shall be called.
- Create a class by clicking on *class*:* or *Browse...* to find an already implemented XPath class. How to create an XPath class will be explained in Section 5.20.3.2: Implementing Custom XPath Functions.

Tip

Reserved namespaces are: as, bit, ecu, lic, node, num, text, var. These namespaces *must NOT* be entered, as custom namespaces with one of these values will be ignored.



5.20.3.2. Implementing Custom XPath Functions

A class that inherits custom XPath functions must extend the `dreisoft.tresos.datamodel2.api.model.xpath.AbstractXPathFunctions`. This class provides helper method for the conversion of Objects or to get return-values that are stuitable for the JXPath engine.

Each `public static` method in the registered class is accessible in an XPath expression by calling `<namespace>:FunctionName`. Each parameter or return-value must be of the type `Object`! Each XPath function must throw an APIInvalidOperationException.

The following example is taken from the test.tresos.plugin.xpath.MyXPathFunctionClass.

```
public class MyXPathFunctionClass extends AbstractXPathFunctions {
```

```
  public MyXPathFunctionClass() {
  }

  /*
   * This function is used in the MyTestModule.xdm to execute an enable check of a node.
   * See the Examples/Float node
   */
  public static Object MyTestXPathFunction( CurrentContext context, Object o )
        throws APIInvalidOperationException {
      DCtxt ctxt = context.getDCtxt();
      DCtxt node = toDCtxt( o );
      if( node.var.getBool() ) {
          return getReturnValue( context, true );
      } else {
          return getReturnValue( context, false );
      }
  }
}
```

The example's method signature means, that there is one function registered for the namespace demo, accessible through demo:MyTestXPathFunction(String xpath). As you might have noticed there is a CurrentContext passed to the function as well. This context is created by the XPath engine itself and does not mean that it has to be passed by the caller. So MyTestXPathFunction only expects only *one* parameter. As the implementer knows which type the parameters should have, they can be converted to the correct Object type using the helper methods:

- toDCtxt(Object param)
- toInt(Object param)
- toFloat(Object param)
- toString(Object param)
- etc. ...

The DCtxt on which the function was initially called can be acquired by calling `context.getDCtxt()`.

To return a value, the method

`getReturnValue( CurrentContext context, Object returnValue )`

should be used.

5.21. Guided configuration API

5.21.1. Purpose

The Guided Configuration API provides the possibility to enhance EB tresos Studio with your own dialogs or editors. With the help of the Guided Configuration API you may create:

- automatic value calculation wizards, e.g. to calculate the Handle-Id's or to recalculate specific values,
- custom module editors to simplify the configuration of a module,
- wizards to simplify often conducted configuration steps, also e.g. along several modules.

5.21.2. Knowledge required

In order to work with the instructions in this chapter, you need to have knowledge of the following topics:

`XML`
: The data of the guided configuration wizards is stored persistently in XML files.

`DCtxt`
: The configuration data of a module is stored in the generic DataModel which can be accessed via DCtxt Public API (Section 5.5.3: Documentation). If you want to exchange data between the generic DataModel and the wizard, then you need to know how to use the DCtxt Public API.

5.21.3. Plug-ins required

After you set up Eclipse as described in Section 5.18: Setting Up Eclipse, you need to create a plug-in project and then add the following plug-ins as dependencies to the `MANIFEST.MF`:

- `dreisoft.tresos.guidedconfig.api.plugin`
- `dreisoft.tresos.lib2.api.plugin`
- `org.eclipse.core.runtime.compatibility`
- `org.eclipse.swt`

Additional optional plug-ins:

- `dreisoft.tresos.datamodel2.api.plugin` if you want to use the DCtxt Public API

To add these plug-ins as dependencies:

- open the `plugin.xml` or `MANIFEST.MF` within Eclipse.

  A special editor opens, the *Plug-in Manifest Editor*.
- Select the tab *Dependencies*.
- Add all plug-ins by using the **Add...** button.

5.21.4. Design overview

This chapter gives a short overview of the design of the guided configuration framework. The following text will introduce new terms to you. All new terms are written in italics. You can find a table of these new terms with an explanation in Table 5.51: Overview API.

Each guided configuration wizard is built on the Model-View-Controller pattern.

Model
: The model part is stored in a *Memento* tree.

  The wizards store data persistently in XML files, which are easily accessible through the `Memento` and `MementoOperationHandler` class. For details about the data representation please refer to Section 5.21.4.4: Data representation.

View
: The view part is represented by the *Wizard Page* that consists of one or more pages.

Controller
: The controller part is represented by the *Backend* class that manages the lifecycle of the wizard. Find more information about the lifecycle below and about the methods that control the lifecycle in chapter Section 5.21.4.3: Lifecycle

There exist four different kinds of wizards. Each wizard type has one place inside of EB tresos Studio from where it can be opened. These places are called *docking points*. For more information about the different wizard types, refer to Section 5.21.4.1: Wizard type overview

A wizard always depends on a project. To restrict whether a wizard is to be available for a specific project, the *Selection Context* can be used. How to restrict the visibility of a wizard, please refer to Section 5.21.7.3: Restricting the visibility of a wizard

When you open the guided configuration wizard, EB tresos Studio displays the GUI with all the *Widgets*. The data, stored persistently in XML files, is loaded and displayed in the GUI. Furthermore the state of the widgets (e.g. the column width of a Table) is loaded and the widgets are adapted.

The Guided Configuration API provides a collection of basic GUI widgets that can be created via the *WidgetFactory* class. If these widgets don't meet your requirements, you can create and use own custom widgets.

The GUI validates the data after each data change and displays markers for errors and warnings. You may navigate to the errornous parameter via a click on the error message.

When you close the wizard, the `Backend` class validates the data. If no errors are reported, the wizard is triggered to run.

Then the data is written to an XML file. Optionally, you may store data in the DataModel by using the *PushService*. For details about the PushService, refer to Section 5.21.4.6: PushService. Access to the generic configuration DataModel is provided through a DataModel API, which writes data directly to the configuration tree.

You can add your own *ResultWidget* to display users the results of the wizard run, e.g. which data has been changed.

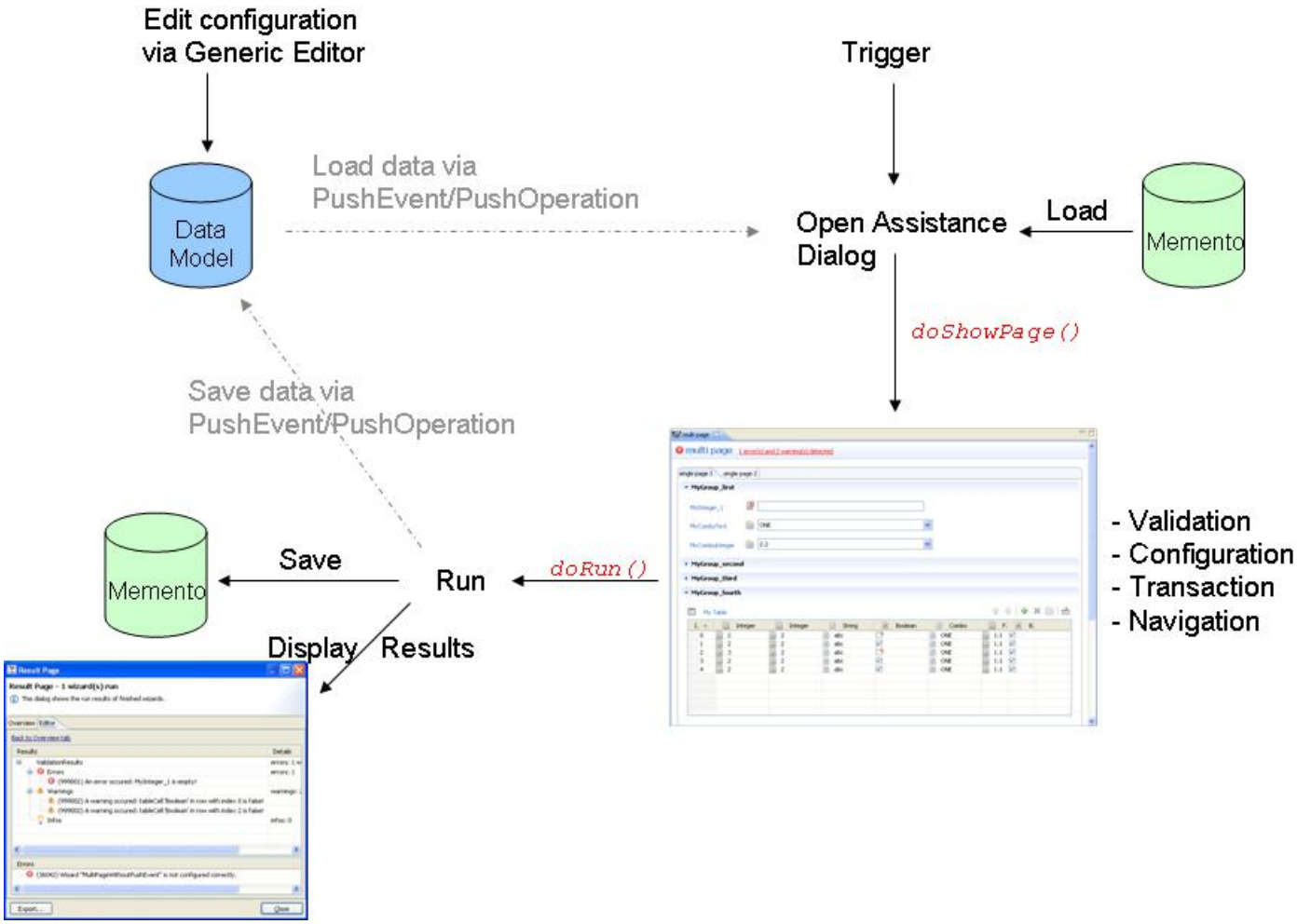


Figure 5.78. Guided configuration workflow

The wise owl is the icon of the guided configuration

The characteristic icon of the guided configuration wizards is the wise owl .

The following terms are important before you read further on:

Term	Description
Backend	The Backend extends the `AbstractBackend` class and handles the background functionality of a wizard. As in the Model-View-Controller concept it corresponds to the controller. It manages the lifecycle of the guided configuration wizard. For more information about the lifecyle of a wizard, refer to Section 5.21.4.3: Lifecycle.
Selection Context	There exist several classes (`ECUConfigContext`, `ModuleConfigContext` and `SystemConfigContext`) providing information about the currently selected project and module. The visibility of docking points in the GUI or execution of *PushOperations* can be restricted with help of the *Selection Context*. You find more details about the Selection Context in Section 5.21.4.2: Selection Context.
Memento	The Memento corresponds to the model in the Model-View-Controller concept as it is the data representation of the guided configuration wizard. Two classes provide access to the guided configuration data layer, `Memento` and `MementoOperationHandler`. Refer to Section 5.21.4.4: Data representation for information when to use which class.
Wizard Page	A Wizard Page is the viewable part of a wizard and can consist of one or more pages. Wizard Pages can be displayed in an editor or a dialog. You can either ▶ create Wizard Pages coding in Java extending the classes `AbstractPage` or `AbstractMultiPage`. Refer to Section 5.21.7.5: Creating the page. ▶ or define the Wizard Pages using the XForms page description. Therefore you register the `XFormMultiPage`. For more information refer to Section 5.21.7.4: Using XForms as GUI description.
Widget	A GUI element for displaying values, that has its own data representation. The Guided Configuration API provides a set of basic gui widgets, for more information refer to Section 5.21.4.5.5: Predefined widgets
WidgetFactory	The `WidgetFactory` class contains methods for the creation of GUI widgets and also a method to register custom widgets.
PushService	The Guided Configuration API provides a pattern called *PushService* to encapsulate the data exchange between the data of the guided configuration wizard (Memento) and the DataModel configuration. The PushService consists of *PushEvents* and *PushOperations*. The Backend implementation sends a PushEvent and the corresponding PushOperation is executed. The operations are used to exchange data between the generic DataModel and the Mementos. For details about PushEvents and PushOperations, refer to Section 5.21.4.6: PushService.
PushEvent	A PushEvent is sent to start a specific PushOperation. The PushEvent is bidirectional and can return the result from the PushOperation to the Backend.

Term	Description
PushOperation	The *PushOperation* is responsible for the data transfer from the wizard to the datamodel and back. A *PushOperation* can be reused for more than one wizard, by passing a specific event to which it listens.
ResultWidget	The Guided Configuration API provides the possibility to register a `ResultWidget`, that displays the result of one ore more wizards in a well formatted way. (Section 5.21.4.5.7: ResultWidget)

Table 5.51. Overview API

5.21.4.1. Wizard type overview

Several places exist inside of the EB tresos Studio from where a wizard can be opened. These places are called *docking points*. The following docking points are available:

- Project Explorer
- Autoconfiguration dialog
- Sidebar view

To be able to use a wizard, you have to define and register two parts:

- Section 5.21.7.1: Registering a wizard, with its GUI representation and lifecycle.
- Section 5.21.7.2: Registering the docking point, where the wizard shall appear in the EB tresos Studio GUI.

Therefore it is possible that several docking points share the same wizard registration.

Based on the docking points, the Guided Configuration API distinguishes different kinds of wizards, namely:

- Sidebar wizard, which can appear as
  - dialog: Section 5.21.4.1.2: Dialog
  - or editor: Section 5.21.4.1.1: Stand-alone editor
- Section 5.21.4.1.3: Autoconfiguration wizard
- Section 5.21.4.1.4: Custom module editor

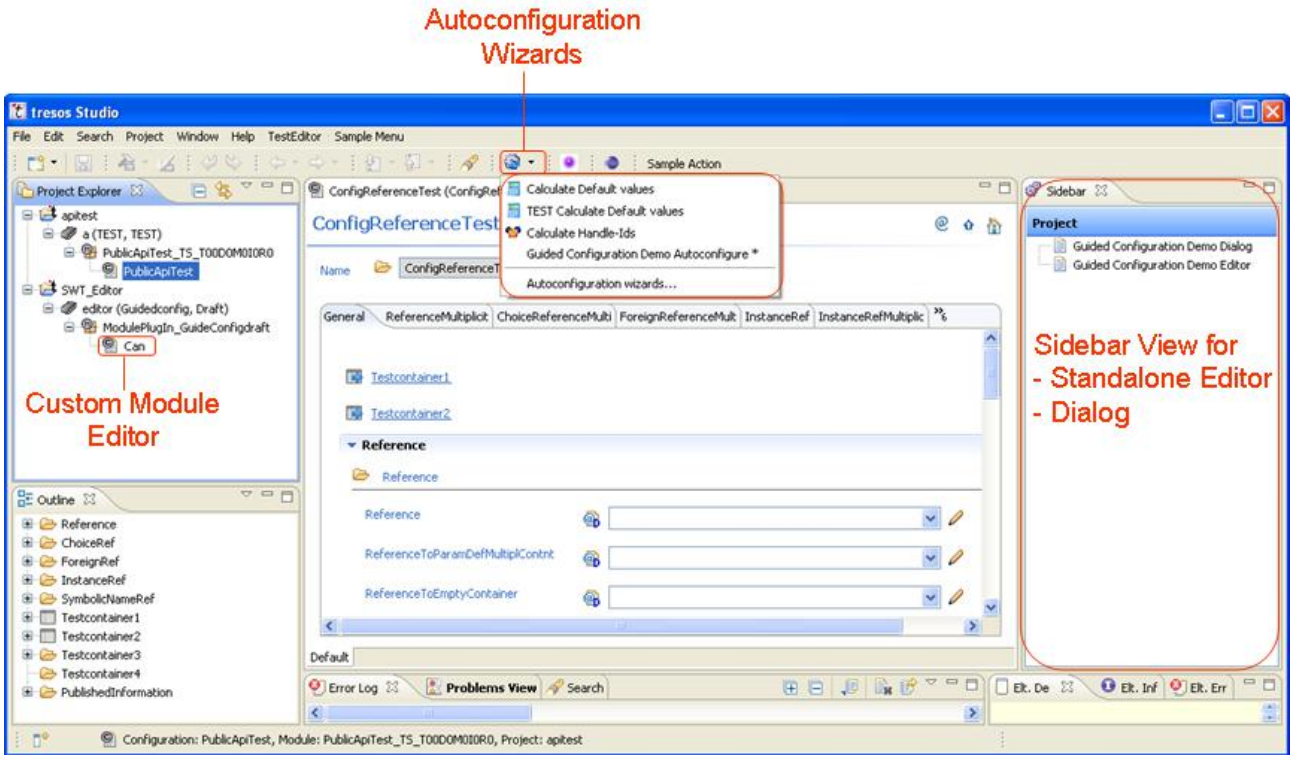


Figure 5.79. Guided configuration wizard type overview

Note

The docking points in the GUI into the *Sidebar* view and the Autoconfiguration dialog are registered via the Triggerextension point `dreisoft.tresos.guidedconfig.api.plugin.trigger`. Thus these docking points are also called "Trigger".

5.21.4.1.1. Stand-alone editor

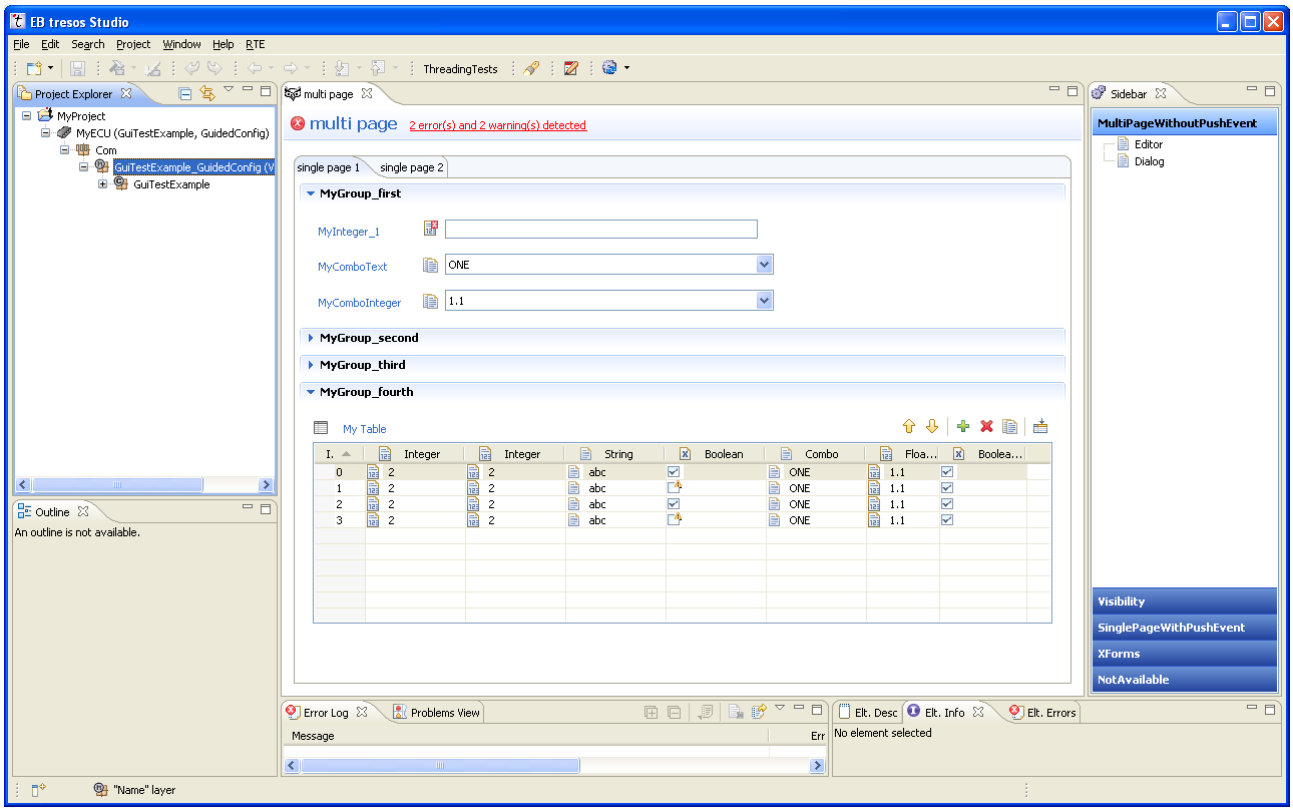


Figure 5.80. Stand-alone editor

Docking point

The stand-alone editor is only available in the *Sidebar* view.

Error handling

Errors and warnings are displayed at top of the editor as a hyperlink. The hyperlink summarizes how many errors and warnings appear in the wizard. Clicking on the hyperlink opens a list of all errors and warnings. It is possible to navigate to each errornous parameter by selecting an error or warning from the list.

Undo/Redo support

Users are able to use the undo/redo functionality.

Multi-page support

Users may switch between pages by clicking into another tab page in the respective editor.

Running the wizard

Closing the editor triggers the wizard to run. If the wizard has errors, the wizard cannot run and the user is asked if he wants to close the editor and loose its data, or to return to the wizard to change the configuration. The results of the wizard run are displayed in a result dialog.

5.21.4.1.2. Dialog

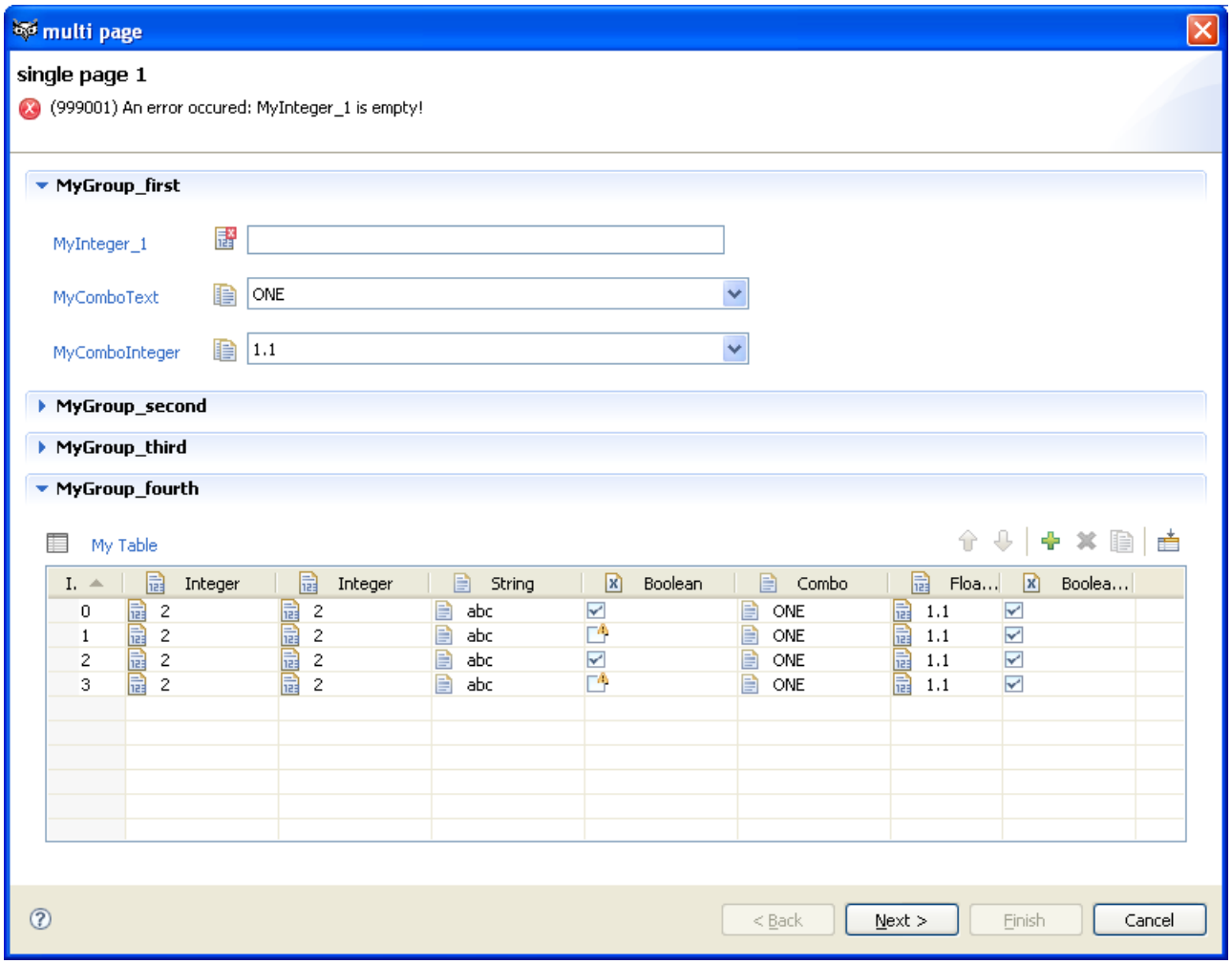


Figure 5.81. Dialog

Docking point

The modal dialog is only available in the *Sidebar* view.

Error handling

If the wizard has errors or warnings, only one item is displayed at a time on top of the page. As soon as the displayed error is fixed, the next error/warning of the list is displayed. First all errors are displayed. If no errors exist anymore, the warnings are displayed one by one. It is not possible to navigate to the parameter by clicking on the error message.

Undo/Redo support

The dialog does not provide Undo/Redo support.

Multi-page support

Users can switch from one page to another by using the **Back** and **Next** buttons.

Running the wizard

The dialog can be finished via the **Finish** button. This button is only enabled if the wizard has no errors. After pressing the **Cancel** button a message is displayed, which asks users if they want to save the changes. The results of the wizard run are displayed in a result dialog.

5.21.4.1.3. Autoconfiguration wizard

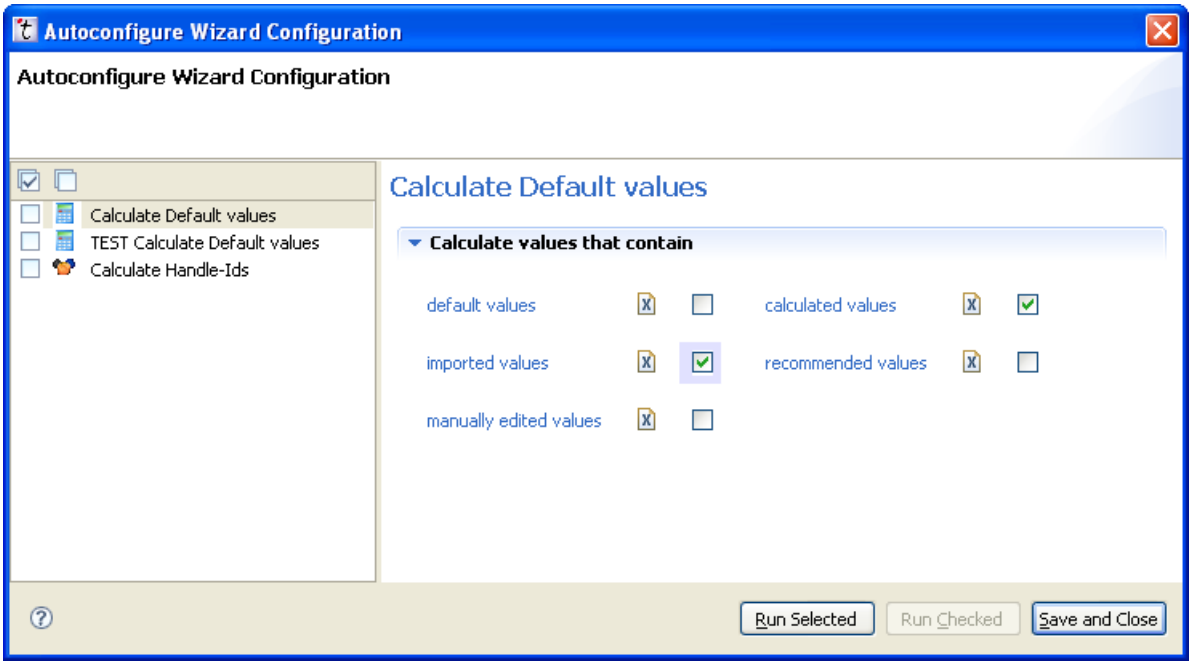


Figure 5.82. Autoconfiguration wizard

Docking point

The autoconfiguration wizard is only available in the *Autoconfiguration* dialog.

Error handling

If the wizard has errors or warnings, only one of these is displayed on top of the page. As soon as the displayed error is fixed, the next error is displayed. It is not possible to navigate to the parameter via the error.

Undo/Redo support

The dialog does not provide undo/redo support

Multi-page support

The user can switch from one page to another by clicking into the tab of another page in the right part of the autoconfiguration dialog.

Running the wizard

An autoconfiguration wizard can be run without user interaction, if it was configured correctly in the GUI once. It can be run either from the GUI and also from the commandline. If run from the GUI the results of the wizard run are displayed in a result dialog.

The GUI provides several possibilities to run the wizard:

- inside the autoconfiguration dialog use the buttons **Run selected** and **Run Checked**. Each checked wizard will appear with a '*' in the list below the gear button .
- click on the gear icon and select one item from the list

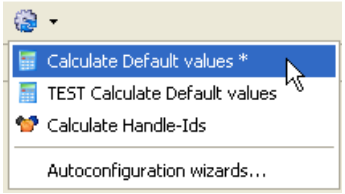


Figure 5.83. Autoconfiguration toolbar

- after checking the wizards in the autoconfiguration dialog, just click on the gear button and all wizards with '*' are run.

The autoconfiguration Trigger can also be executed via Commandline in addition to the Gui.

```
tresos_cmd.bat autoconfigure <projectname> [<triggerid>,...]
```

If only the projectname is specified, then all available and checked autoconfiguration triggers for the given project will be executed. There is also the possibility to restrict the execution by specifying a comma separated list of triggers. This list is checked one after the other. The trigger is only executed, if it exists and if the visibility restrictions for the given project are fulfilled.

5.21.4.1.4. Custom module editor

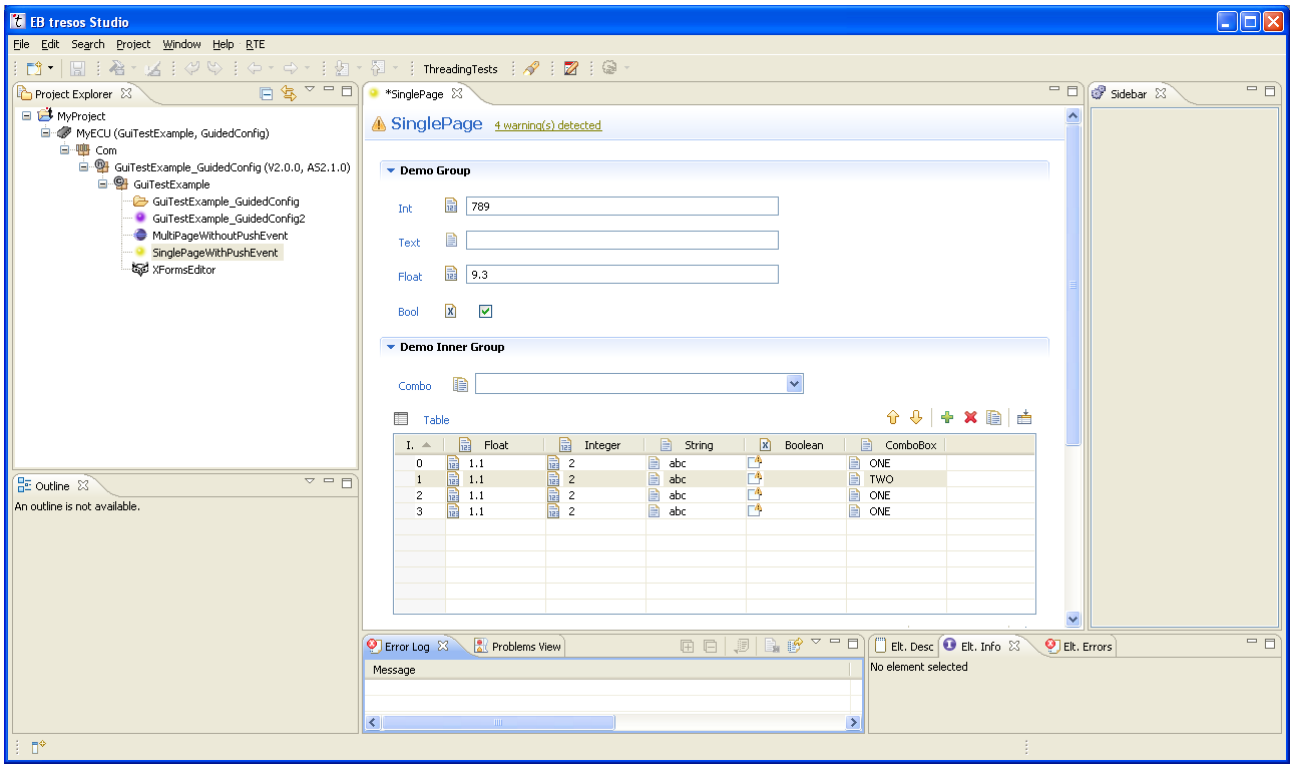


Figure 5.84. Custom module editor

Docking point

The custom module editor is displayed in the same way as the generic configuration editor in the *Project Explorer*.

Error handling

Errors and warnings are displayed at top of the editor as a hyperlink. The hyperlink summarizes how many errors and warnings appear in the wizard. Clicking on the hyperlink opens a list of all errors and warnings. It is possible to navigate to each errornous parameter by selecting an error or warning from the list.

Undo/Redo support

Users can use the undo/redo functionality.

Multi-page support

Users can switch between multiple pages by clicking into the tab of another page in the editor part.

Running the wizard

Closing the editor triggers the wizard to run. If the wizard has errors, the wizard cannot run and users are asked if they want to close the editor and loose its data, or to return to the wizard to change the configuration. The results of the wizard run are displayed in the EB tresos Studio *Error Log* view.

5.21.4.2. Selection Context

You may restrict the visibility of wizards that are registered via extension point `dreisoft.tresos.guided-config.api.plugin.trigger`, aka. trigger. This can be done depending on the current *Selection Context*. To run a wizard you always need a loaded project.

To find out how to restrict the visibility of a wizard, refer to Section 5.21.7.3: Restricting the visibility of a wizard.

There exist several classes, that provide information about the currently selected project and - if applicable - about the selected module.

The following context classes are available:

- `ECUConfigContext` provides information about the currently selected project. If no project is selected or the selected project is not loaded this context is disabled.
- `ModuleConfigContext` provides information about the currently selected module. If no information is available, this context is disabled.
- `SystemConfigContext` is enabled if the currently selected project has a system model.

Each context class also provides the possibility to check if the current context is available at the moment, so e.g. for the `ECUConfigContext` you may check if a loaded project is selected.

The following sections provide you with information about the context classes and the data that you can use to restrict the visibility of the docking points and the runnability of *PushOperations*:

- Section 5.21.4.2.1: ECUConfigContext
- Section 5.21.4.2.2: ModuleConfigContext
- Section 5.21.4.2.3: SystemConfigContext

The contexts are available via the `ContextManager` class. Each context has a unique Id and can be queried by using:

```
ContextManager.getInstance().getContext(ECUConfigContext.ID)
```

When opening a wizard all contexts are cloned and given to the `Backend` class, so the information from which project the wizard is opened does not get lost.

You can query the list of the cloned contexts by calling the method `getContexts()` of the `Backend` class.

5.21.4.2.1. ECUConfigContext

The `ECUConfigContext` class provides information about the current selected project.

This context provides the following data:

- `projectname`: the name of the currently selected project
- `target`: the target of the project (e.g. TRICORE)

- `derivate`: the derivate (e.g. TC1766)
- `relVersion`: the release version (e.g. 2.1, 2.*, 2.1.0,2.1.*)
- `ecuid`: the ecu Id
- `generationPath`: the generation path set in the project

The ECUConfigContext provides information about the modules in the selected project in addition: For each module of the currently selected project the following attributes are provided:

- `moduleId.<ModuleName>` : with value TRUE if at least one module configuration with this Id is available and enabled, e.g. `attribute:moduleId.Can_TS_T16D6M2I1R0;value=TRUE`
- `moduleType.<TypeName>` : with value TRUE if at least one module configuration with this type is available and enabled, e.g. `attribute:moduleType.Can;value=TRUE`

5.21.4.2.2. ModuleConfigContext

The ModuleConfigContext provides information about the modules of the currently selected project. The ModuleConfigContext provides the following data.

- `schemapath`: the schema path of the currently selected node in the editor or empty if none is selected.
- `datapath`: the data path of the currently selected node in the editor or empty if none is selected
- `editorid`: the ID of the currently selected editor
- `moduleid`: the module-ID
- `configurationid`: the configuration-ID
- `dctxt`: the DCtxt of the currently selected node
- `type`: the type of the module (e.g. `PduR`)
- `category`: the module category (e.g. Com)
- `layer`: the module layer (e.g. Service)
- `specVersion`: the module spec version (e.g. 2.0.1, 2.*, 2.1)
- `swVersion`: the software version of the module (e.g. 2.1.1,2.*,2.1)
- `relVersion`: the release version of the module (e.g. 2.1.1,2.*,2.1)

5.21.4.2.3. SystemConfigContext

The SystemConfigContext provides information if the currently selected project has system model support. The data can be used to restrict the visibility of the triggers and the runnability of PushOperations. The SystemConfigContext provides the following data.

- `projectname`: the name of the currently selected project

5.21.4.3. Lifecycle

The lifecycle of a wizard is represented by the `Backend` class. At special times in this lifecycle you can modify the wizard behaviour by overwriting specific methods in your Backend implementation.

The following figure shows the timeline of the lifecycle methods, that can be overridden in you `Backend` class.

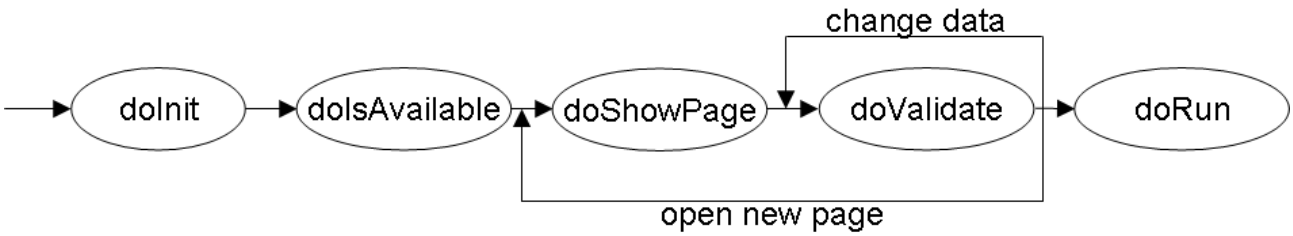


Figure 5.85. Wizard lifecycle

Method	Description
`doInit()`	Implement this method to initialize your `Backend` class. `doInit()` is called right after instantiation of the Backend before anything else is initialized. This means that the root Memento has not yet been created and there is no GUI. This method is only called once in the wizard lifecycle.
`doIsAvailable()`	Implement this method if there are restrictions under which the wizard cannot be displayed. You need to return the reason why the wizard cannot be opened. Return the reason as `APIOperationStatus`. The return code is then displayed to the user. This method is called once right before the wizard is opened.
`doShowPage()`	This method is called each time a new page is displayed. For multi-pages this method is also called for each page switch. Implement this method to load the data and fill it into the GUI data representation.
`doValidate()`	Implement this method to validate the data so the validation results are sent to the GUI in order for the error and warning markers to appear. This method is called when the wizard is opened and before it is finished and each time a value is changed.
`doRun()`	This method does the work for this wizard and shall be implemented to save the data to the datamodel. It is called once when the wizard is run. The wizard is only run, if no errors are returned by `doValidate()`. If there are errors, this method is not executed.

Table 5.52. Backend lifecycle methods

5.21.4.4. Data representation

Each wizard has its own set of data, in which each widget is connected to one element. The data is organized as a tree which is stored persistently in XML files, located in the project workspace in the subfolder `.prefs` as `.mem` files. The runtime representation of the data is a tree of Memento objects.

Several ways exist to manipulate the data. The recommended way to change the Memento tree is to use the `MementoOperationHandler` class. This encapsulates the direct access via the `Memento` class. The `MementoOperationHandler` class provides undo/redo support and transaction handling. For details about the `MementoOperationHandler` class, refer to Section 5.21.4.4.1: The MementoOperationHandler.

The content of a `.mem` file looks like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <memento version="1.0" lastmodified="Aug 7, 2009 10:34:13 AM">
   <guidedDemo4 comboText="TWO" text="3453" float="3453.0" comboNumber="1.1">
     <__metadata>
       <Group_1_0 __expanded="true"/>
       <text __radix="DEC" __separators="false"/>
       <comboText __radix="DEC" __separators="false"/>
       <table __width="50 100 100 100 100 95" __visible="yyyyyy"/>
       <Tree __width="136 100 100 100 100" __visible="yyyyy"/>
            ....
     </__metadata>
     <table>
       <__ IntColumnMemento="2" FloatColumnMemento="1.1"
         StringColumnMemento="abc" BooleanColumnMemento="false" ComboBoxMemento="ONE"/>
       <__ IntColumnMemento="2" FloatColumnMemento="1.1"
         StringColumnMemento="abc" BooleanColumnMemento="false" ComboBoxMemento="ONE"/>
              .....
     </table>
     <Tree>
       <__ DescMemento="1.1" IntColumnMemento="2"
         StringColumnMemento="basic" BooleanColumnMemento="true" ComboBoxMemento="ONE">
         <__ DescMemento="1.1" IntColumnMemento="2"
           StringColumnMemento="abc"
           BooleanColumnMemento="false" ComboBoxMemento="ONE">
           <__ DescMemento="1.1" IntColumnMemento="2"
             StringColumnMemento="abc"
             BooleanColumnMemento="true" ComboBoxMemento="ONE"/>
         </__>
     </Tree>
   </guidedDemo4>
```

```
</memento>
```

The XML tag *guidedDemo4* represents the root Memento and is named after the wizard-ID. All widgets with single values are represented by attributes.

More complex widgets are connected to a sub tag of the root tag. The specific data representation depends on the widget. At the moment the WidgetFactory provides only two widgets that are connected to a sub tag:

Table
: The sub tag for the Table contains a list of row entries. Each cell value is represented by an attribute of the row tag.

TreeTable
: The sub tag of the TreeTable contains a tree of nested elements.

In addition to the user data a node for the meta data exists, like:

- width of Table columns
- expand state of groups
- radix settings of integer parameters

5.21.4.4.1. The MementoOperationHandler

The MementoOperationHandler class provides access to the data representation. It provides getter- and setter-methods for Memento values, as well as methods to manipulate special widget, e.g. to create a new row for a Table.

This is the main class to perform Memento operations and the recommended way working with the data. Using this class provides undo/redo support and transaction handling.

Transactions shall be used to combine several actions into one atomic action, so the user can undo these actions with one click.

The MementoOperationHandler provides methods for:

- setting values of different types(String, boolean, int, float)
- adding and deleting Mementos
- Table operations (addRow, moveRow, removeRows, duplicateRows)
- TreeTable operations (addTreeRow, removeTreeRows, duplicateTreeRows)

Find the exact syntax of the methods in the java documentation, see Section 5.21.5: API.

Note

Query the MementoOperationHandler by using

`backend.getMementoOperationHandler().`

5.21.4.4.2. The Memento Interface

The `Memento` class is the data layer access of the Guided Configuration Public API. It provides getter- and setter-methods for Memento values, as well as methods to create new Mementos. The underlying data has a hierarchical structure which is persistently stored in XML format. It has a top node (the Table itself), a subnode for each row, and at least one attribute for each cell entry.

5.21.4.4.3. MementoFactory

The MementoFactory defines the construction of the Memento tree. It is comparable to a Schema file in XML. The MementoFactory is used to

- set default values
- calculate specific values
- define which Memento is to be persistent or not.

A custom factory must extend `AbstractMementoFactory` and implement only one method `getMemento( Memento parent, String name ).`

Note

The MementoFactory has to be a singleton class!

5.21.4.5. GUI abstraction

The guided configuration framework provides common functionality for the GUI representation. The following elements are provided:

- Section 5.21.4.5.1: Multi-page support
- Section 5.21.4.5.2: Layouting
- Section 5.21.4.5.3: Validating
- Section 5.21.4.5.4: Navigating
- Section 5.21.4.5.5: Predefined widgets

- Section 5.21.4.5.6: Widget lifecycle
- Section 5.21.4.5.7: ResultWidget

5.21.4.5.1. Multi-page support

A wizard can consist of one or more pages. The navigation between the pages is done by the guided configuration framework. Depending on the kind of wizard, the navigation between multiple pages is done either via tab pages or via **Back/Next** buttons. For an explanation which wizard uses which kind of page navigation please refer to Section 5.21.4.1: Wizard type overview.

5.21.4.5.2. Layouting

Each page is based on a GridLayout built on 4 columns. Part of the widgets span several columns to provide a nice layout.

To find out which widget uses which span width, refer to Section 5.21.4.5.5: Predefined widgets

5.21.4.5.3. Validating

You only need to implement the validation of the data in the Backend `doValidate()` method. The guided configuration framework will update the error and warning markers in the GUI corresponding to the results of the validation. Each data change will trigger a validation and update of the GUI.

The validation of the data is run asynchronously, so as not to disturb the user who edits the page.

The errors and warnings are displayed on top of the page; either only one (for dialogs and autoconfiguration wizards) or all as hyperlinks (for stand-alone editors and custom module editors). The hyperlinks can directly navigate to the errornous widget, even if the widget is on another page inside the wizard. For an example see Section 5.21.6.2: Demo1

5.21.4.5.4. Navigating

The GUI abstraction layer also provides the possibility to add your own navigation links to a page. There exists a predefined browser widget that can contain hyperlinks. With these links it is possible to navigate to a specific widget inside a wizard.

Please refer to the following example code. It shows how to navigate to a specific widget inside the wizard:

```
<a href="wizard://<wizard-ID>/<widget-ID>">
```

Where wizard-ID is the Id defined in the `dreisoft.tresos.guidedconfig.api.plugin.guidedconfig-wizard` extension point. And the widget-ID is the unique Id of the widget to navigate to.

5.21.4.5.5. Predefined widgets

All widgets provided by the Guided Configuration API are available via the `WidgetFactory` class.

The WidgetFactory class provides methods for the simple creation of GUI widgets. It connects each widget to the underlying data layer (Memento), by registering it with a user-chosen Memento-ID. It also defines the layout for the widgets. The standard layout is based on a GridLayout using four columns.

To be able to use the WidgetFactory within the `doCreateControls()` method, just call `getWidgetFactory()` and you'll get the instance for this page. The WidgetFactory provides the following collection of basic widgets:

Widget type	Description
Group	A grouping composite. If you use frame type `TITLE`, then the group can be collapsed. It spans all four columns in the layout . Group1
Label	A text label. It uses one column in the layout. Combo Text
CheckBox	A checkbox that is of the type boolean. It uses one column in the layout and has type icon .
Button	A button that can either have text or an icon as label. It uses four columns in the layout. The button can have the following styles: - `ButtonFlags.PUSH` - `ButtonFlags.TOGGLE` FilterTable .
Text	A plain textfield with a type icon. A textbox can use the following types: - `TextFormat.TEXT` - `TextFormat.INTEGER` - `TextFormat.FLOAT` It uses three columns in the layout . 3453
ComboBox	A drop-down-box that can use the following types: - `TextFormat.TEXT` - `TextFormat.INTEGER` - `TextFormat.FLOAT` It uses three columns in the layout and has type icon 1.1
Browser	A browser part that can contain HTML content with hyperlinks. Users can choose how many columns of the GUI it spans.

Widget type	Description
Spacer	A placeholder widget. It does not display anything, but only uses space within the GUI for layouting purposes. The spacer can span several columns. So for example users can define, that one row displays just one label and a checkbox instead of two, which is standard.

Table	A table with several user defined columns and one index column. The user defined columns are configured by the `SWTColumnDescription` class. The Table itself needs a Memento-ID for the top node Memento. The Table spans all four columns of the layout. For each row a sub Memento is created, which contains the values for each column. The path to a specific column value within the Memento would be: `/tableId/rowNumber/columnId` of which
	
	`tableId`
	is the Memento-ID of the Table Memento.
	
	`rowNumber`
	is the Memento-ID for a Table row, where `rowNumber` is the index of the row.
	
	`columnId`
	is the Memento-ID for the column, defined via the `SWTColumnDescription`

Widget type	Description
TreeTable	A combination of a tree and a table. The first column displays a tree (tree column) and the other columns display the data for each tree item. The TreeTable spans four columns of the layout. The tree column is configured by the `SWTTreeColumnDescription` class and uses the type `TextFormat.TREE`. The `SWTTreeColumnDescription` provides the possibility to define iconPaths You may specify an icon for each level. editableRows You may set each row level editable or uneditable. If a row level is uneditable, only the data for the tree column is displayed, all other columns remain empty for this level. structuralEditableRows You may set a row level as not-structurally-editable. This means, that no child items can be added or deleted. The tree column displays the structure of the data. Each tree item consists of an optional icon and a text entry. All other columns are configured by the `SWTColumnDescription` class. The TreeTable itself and each column needs a Memento-ID. The path to locate a specific column value within the TreeTable Memento would be `/treeId/rowPath/columnId`, where `rowPath` is of the form: rowIndex(/rowIndex)* (e.g.: `/treeId/0/2/columnId` represents the Memento path for the column 'columnId' of the third child of the first toplevel item) **Note** It is not possible to add more than 32 columns to the TreeTable.

Table 5.53. WidgetFactory basic widgets

Here an example for the Table and the TreeTable

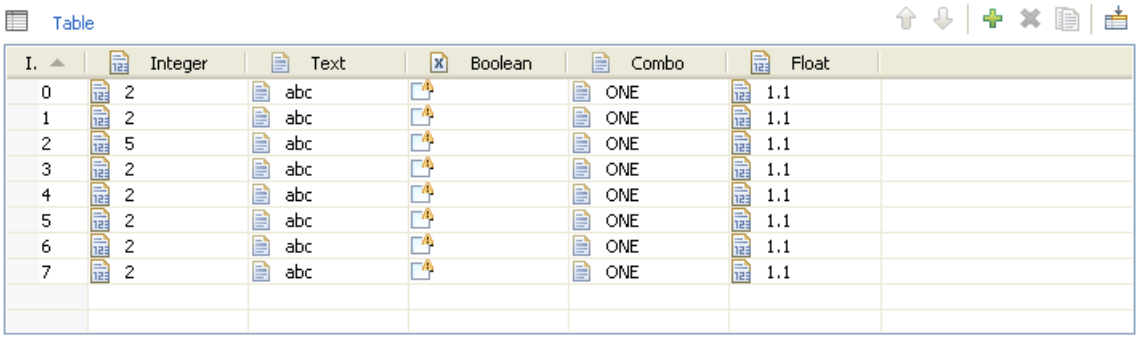


Figure 5.86. Table

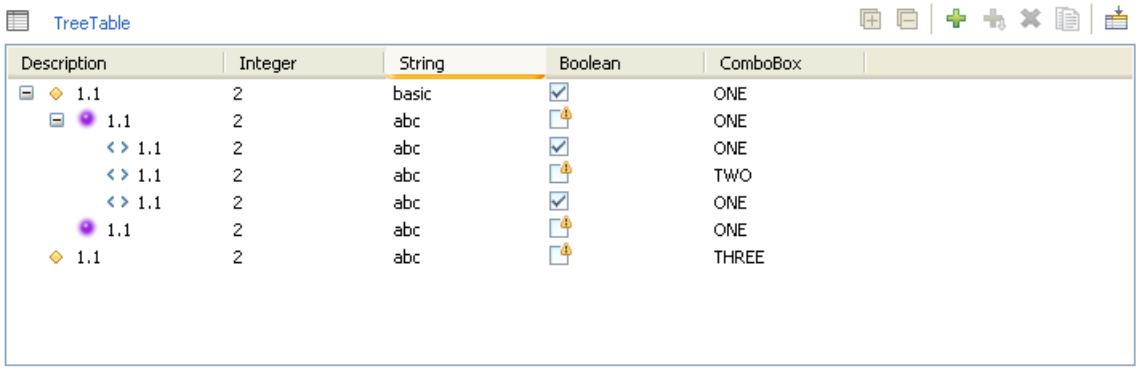


Figure 5.87. TreeTable

Warning

If you create the widgets by using the `WidgetFactory`'s create methods, call `factory.layout(parent)` at the end of the `doCreateControls` method of the Wizard Page.

Tip

If the predefined widgets don't meet your requirements, you can use your own custom widgets. See Section 5.21.7.6: Implementing a custom widget for more information on custom widgets.

Note

If you prefer to write your GUI representation in XML form, see Section 5.21.7.4: Using XForms as GUI description for more information on using XForms.

5.21.4.5.6. Widget lifecycle

The Guided Configuration API also provides methods to manipulate a widget during its lifecycle. The following methods can be overridden to change the data of the widget as well as its appearance.

`AbstractBackend.doHandleUpdatedData()`

This method can be overridden to change GUI elements that depend on the content of another GUI element. It is called when data in the GUI was changed and after the data was written to the Mementos.

`AbstractBackend.doUpdateWidgetEnablement()`

This method can be used to change the enablement of a widget. It is called asynchronously each time after data is changed .

`XFormMultiPage.doAdaptWidget()`

If you use the XForms representation of the GUI, this method can be used to adapt the widget after loading the page. This can be e.g. adding an event listener or Table actions.

5.21.4.5.7. ResultWidget

After running a wizard a result dialog is displayed for autoconfiguration wizards, stand-alone wizards and dialogs.

The result dialog contains several tab pages. The first tab, named *Overview* is always present and displays a summary over all wizards run.

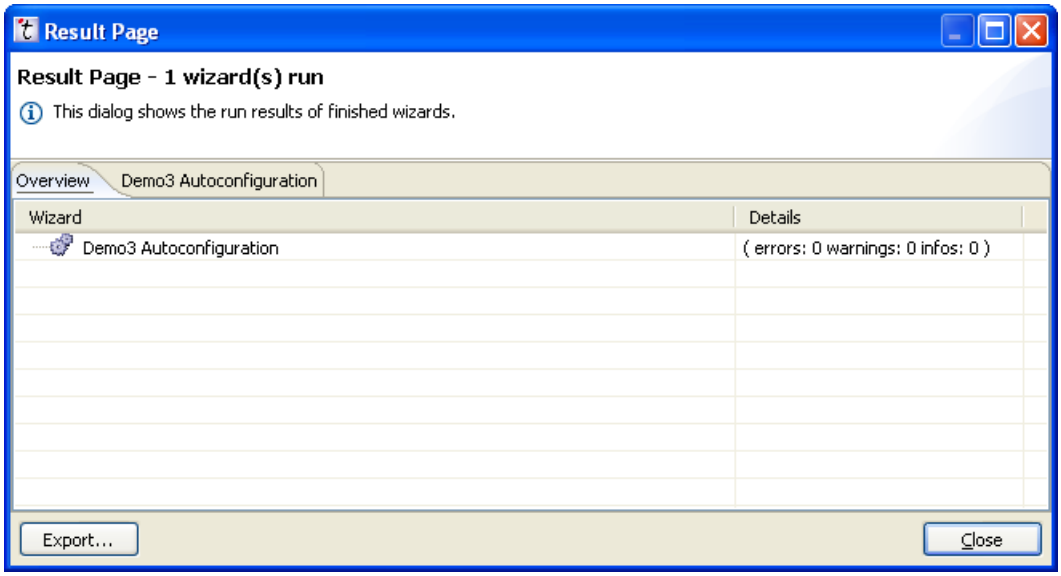


Figure 5.88. Result dialog overview

In addition to the *Overview* tab, there exists one tab page for each wizard run. The tab page displays either your own `ResultWidget`, registered in the `dreisoft.tresos.guidedconfig.api.plugin.guidedconfigwizard` extension point or the `DefaultResultWidget`.

The `DefaultResultWidget` displays the list of `APIOperationStatus` that are returned by the `doRun()` method.

Your own `ResultWidget` can e.g. display which parameters of the datamodel configuration have been changed by the `PushOperation` (see Section 5.21.7.11: Registering a custom ResultWidget).

5.21.4.6. PushService

The PushService consists of a PushEvent and a corresponding PushOperation that listens to the event.

A PushOperation is triggered from within the Backend by sending a PushEvent. The PushOperation is responsible for the data transfer from the wizard to the datamodel and back:

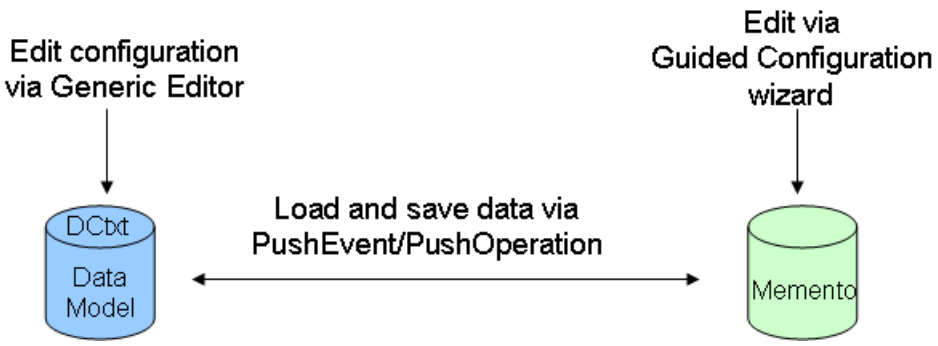


Figure 5.89. PushService data flow

A PushOperation can be reused for more than one wizard, by passing a special kind of event.

A PushEvent is bidirectional and can return the result from the PushOperation to the Backend.

A PushOperation can be registered in two ways:

- via extension point `dreisoft.tresos.guidedconfig.api.plugin.pushservice`:

  here you have to register a class that implements `AbstractConfigurablePushOperation`. You also may restrict the runnability of the PushOperation by using the event tag. You can either restrict the PushOperation to a specific event class or to a specific *Selection Context* by using the `event` tag.

- via `PushService` registry:

  you can further register a PushOperation via `PushService.getInstance().addPushOperation(AbstractPushOperation operation )`. The added operation must implement `AbstractPushOperation`. The extended class can implement an own `doCanHandle( AbstractPushEvent event )` method, to query if the operation can handle the given event.

Note

A PushOperation is *NOT* a must. The Backend can also write to the EB tresos Studio datamodel directly via the Public Api DCtxt class.

5.21.4.7. Locking mechanism

If a wizard for a specific project is displayed, all functions that access the datamodel of *this* project are locked. That means:

- No module can be added or removed from the project .
- A configuration editor can still be opened and browsed, but not edited .
- No generate or verify can be run.
- Importers/Exporters can be created, but not run.
- Just one wizard at a time can be open per project. When you try to open another wizard for the same project a dialog is displayed:

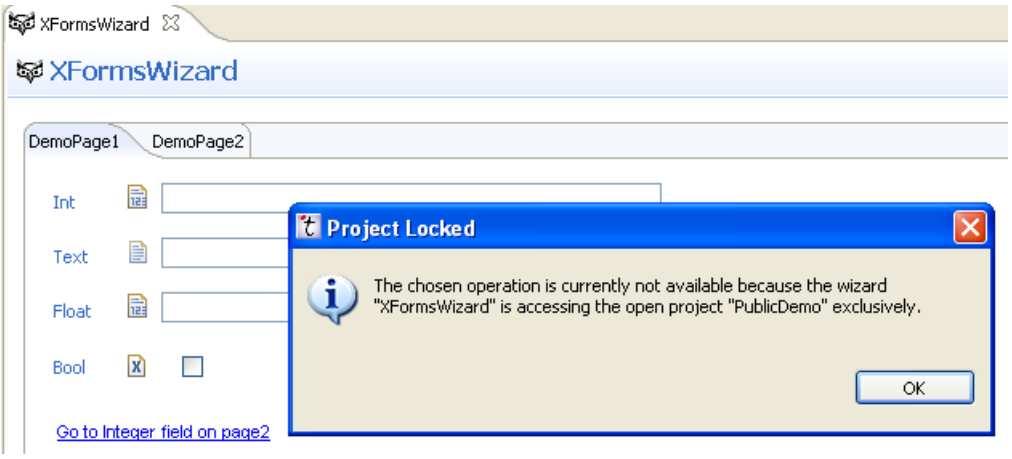


Figure 5.90. Dialog displaying that the current project is already locked.

- The project cannot be saved while the wizard is displayed.

5.21.5. API

Find the exact syntax of the methods in the Java documentation. The Java documentation is located in your EB tresos Studio installation folder under `doc/API` or in the EB tresos Studio online help. To access the online help, open the EB tresos Studio GUI and press **F1**.

Tip

Do not use methods in internal packages or methods marked as *deprecated*.

Tip

Methods with prefix *do* are meant to be overwritten.

5.21.6. Demos

5.21.6.1. Setting up the demo plug-in

This chapter describes several demos and their specifics. To be able to use the demos please follow the instructions below.

A demo plug-in is available under `<tresos-install-loc>/demos/PublicApiTest_TS_T00D0M0I0R0`. Please install this plug-in according to the instruction in Section 5.22: Installing the Public Api Demo Plugin.

Then return to this chapter and follow the instructions below.

After you installed the demo plug-in, create a project with

- target/derivate `TEST/TEST`
- release version 2.1
- and a module `PublicApiTest_TS_T00D0M0I0R0`

If the project is not loaded, load the configuration. You now see several triggers in the *Sidebar* view. If the *Sidebar* view is not visible, then follow the instructions in Section 5.21.6.1.1: Making the Sidebar view visible

The data of the wizards is stored persistently in XML files. You find these files in your project in subfolder `.prefs` in files with extension `.mem`. As default `.* resources` are not shown in the *Project Explorer*. To show the files, follow the instructions in Section 5.21.6.1.2: Customizing the view

Some demos manipulate the DataModel configuration using PushServices. To be able to see the result of the data change, please change the default configuration data. Follow the instructions in Section 5.21.6.1.3: Setting up the data

Now you are done setting up the demo plug-in.

Each demo displays different features of the Guided Configuration API. Find an overview which demo uses which feature in the following list:

	Visibility	Multipage	Custom widget	All predefined widgets	Xforms	Validation	Update widget-enablement	Selection events	Navigation	MementoFactory	PushService	ResultWidget
Standalone editor (demo1)	x	x			x	x			x	x		
Standalone editor (demo2)	x		x				x				x	
Autoconfiguration dialog (demo3)	x										x	x
Custom module editor (demo4)	-			x		x		x				
Dialog (demo5)	x	x				x						
Dialog (demo6)	x											

Figure 5.91. Overview about the features of the demos

5.21.6.1.1. Making the Sidebar view visible

If the *Sidebar* view is not visible in EB tresos Studio, then make it visible via **Window/Show View/Wizard Sidebar**.

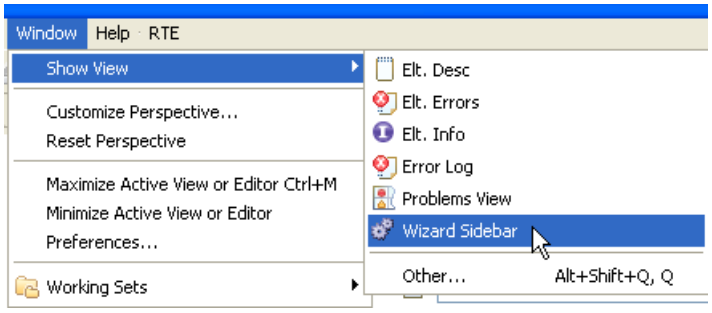


Figure 5.92. Make the Wizard Sidebar visible

5.21.6.1.2. Customizing the view

Please customize your *Project Explorer* view so ".* resources" are displayed.This means you can see the persistent data representation as XML files with extension .mem in the `.prefs` folder of the corresponding project.

To display the .* resources:

- Click on the downward pointing arrow in the *Project Explorer*

  A drop-down menu opens up.

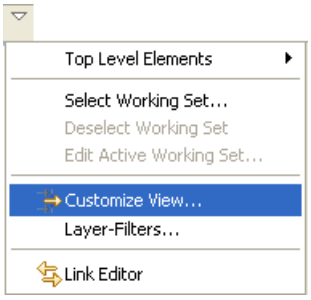


Figure 5.93. Customize your display options

- From the drop-down menu, select the option *Customize View...*.

  The *Available Custiomizations* window opens up.

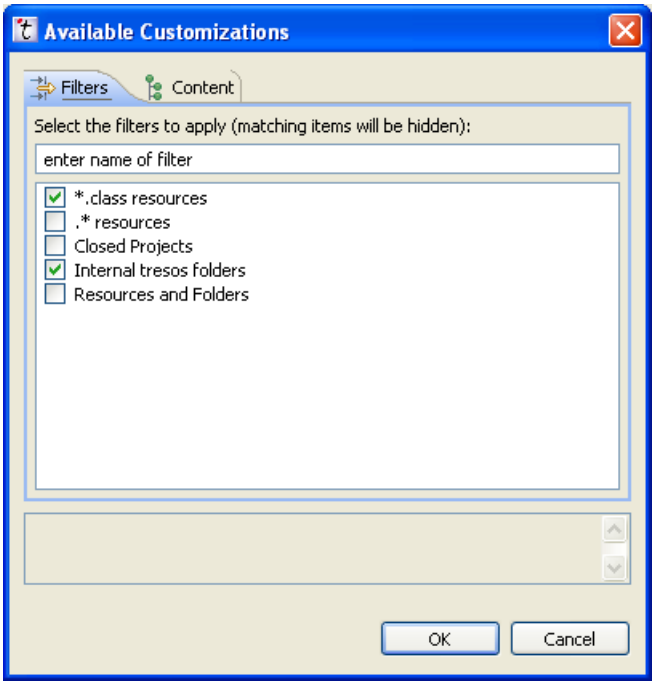


Figure 5.94. *Available Customizations* view

- Uncheck the **.* resources** checkbox.
- Click the **OK** button.

  You have customized the *Project Explorer* to display the persistent data representations as XML files.

5.21.6.1.3. Setting up the data

As precondition for data exchange between the guided configuration wizard and the DataModel,

- please open the generic editor for `PublicApiTest_TS_T00D0M0I0R0` module configuration and navigate to tab page *Stuff* and
- add some values to the configuration parameters.

  This tab page contains the corresponding parameters to the demos for a data exchange between DataModel and guided configuration wizard.

5.21.6.2. Demo1

5.21.6.2.1. Overview

The wizard for demo1 is available in the *Sidebar* view and opens a stand-alone editor in the editor part of EB tresos Studio.

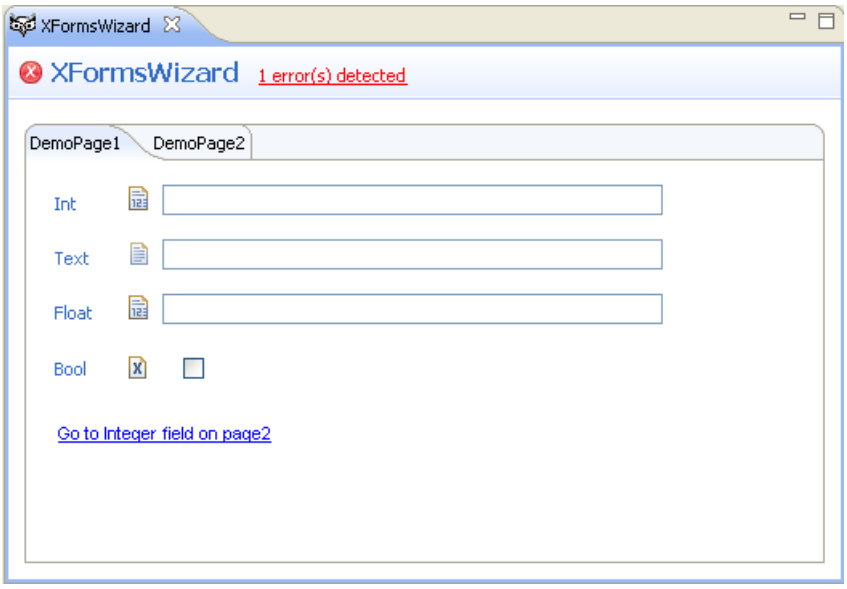


Figure 5.95. Demo1: XForms multi-page

In this example you will learn

- how to restrict the visibility of a wizard, so it is only displayed in the *Sidebar* view if the restriction is fulfilled.
- how multi-pages are displayed in an editor and how to navigate through the wizard
- using an XForms page description to describe the GUI representation in an XML format.
- how the data can be validated and how the validation results are displayed in the GUI.
- how to use own navigation links.
- how to control which data is to be stored persistently.

5.21.6.2.2. Restricting the visibility

This demo Trigger is available in the *Sidebar* view only if a loaded project is selected. This loaded project must have target TEST and a module with type `PublicApiTest`. The visibility restriction is done in the `plugin.xml` that registers the `dreisoft.tresos.guidedconfig.api.plugin.trigger` extension point.

If you disable or remove the module, the Trigger disappears from the *Sidebar* view. As the visibility restriction requires this module.

For more information please refer to Section 5.21.7.3: Restricting the visibility of a wizard.

5.21.6.2.3. Using multi-page support

The wizard has two pages, which are available as tab pages in the editor part of EB tresos Studio. If an error is found it is displayed on top of the page no matter on which page you are. You can switch between the pages by clicking on the tab pages in the editor view of EB tresos Studio.

5.21.6.2.4. Using XForm GUI description

The XForm page description is not written in Java, but is an XML file, that is part of the demo plug-in. You can find this XML file in the demo plug-in at `pages/Pages.xml`. To register an XForms page description, please refer to Section 5.21.7.4: Using XForms as GUI description.

5.21.6.2.5. Validating

After opening the wizard an error notification is displayed on top of the page. This error summary is a hyperlink and can be used to navigate to the errornous parameters. In this case, when you click on the error *(GUIDED-DEMO_1) An error occured: Integer field on page 2 must not be empty*, then tab page *DemoPage2* is activated and the Integer field becomes active.

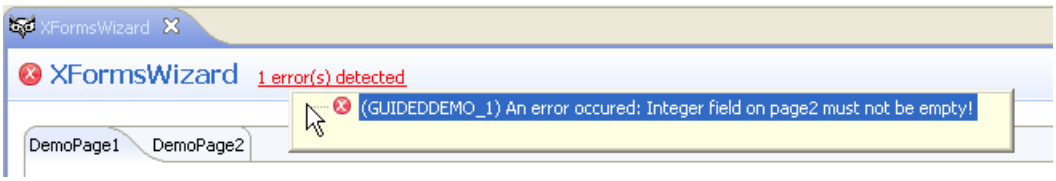


Figure 5.96. Demo1: Validation summary as hyperlink on top of the page

5.21.6.2.6. Navigating

With the browser widget you may add your own hyperlinks to navigate to other widgets. The example code below shows you how to do this:

```
<a href="wizard://guidedDemo1/Page2IntWidget">Go to Integer field on page2</a>
```

The `WidgetFactory` class knows which widget is on which page and navigates to the correct page and sets the focus to the specified widget.

5.21.6.2.7. Using a custom MementoFactory

This demo uses its own `MementoFactory` class instead of the `DefaultMementoFactory` class. Via the MementoFactory you can define which parameters are to be persistent and which not.

So you can store only those parameters which are not again read from the DataModel when the wizard is opened the next time.

In this demo when you insert some values into the parameter, close and reopen the wizard. Now only the values of the following parameters are stored persistently and are available after reopening the wizard:

- Text (DemoPage1)
- Bool (DemoPage1)
- Integer (DemoPage2)

How to create your own `MementoFactory`, refer to Section 5.21.7.9: Creating a MementoFactory

5.21.6.3. Demo2

5.21.6.3.1. Overview

The wizard for demo2 is availabe in the *Sidebar* view. Opening the wizard *Demo2 Custom Widget* shows a stand-alone editor in the editor part of the EB tresos Studio. The speciality of this demo is a custom widget.

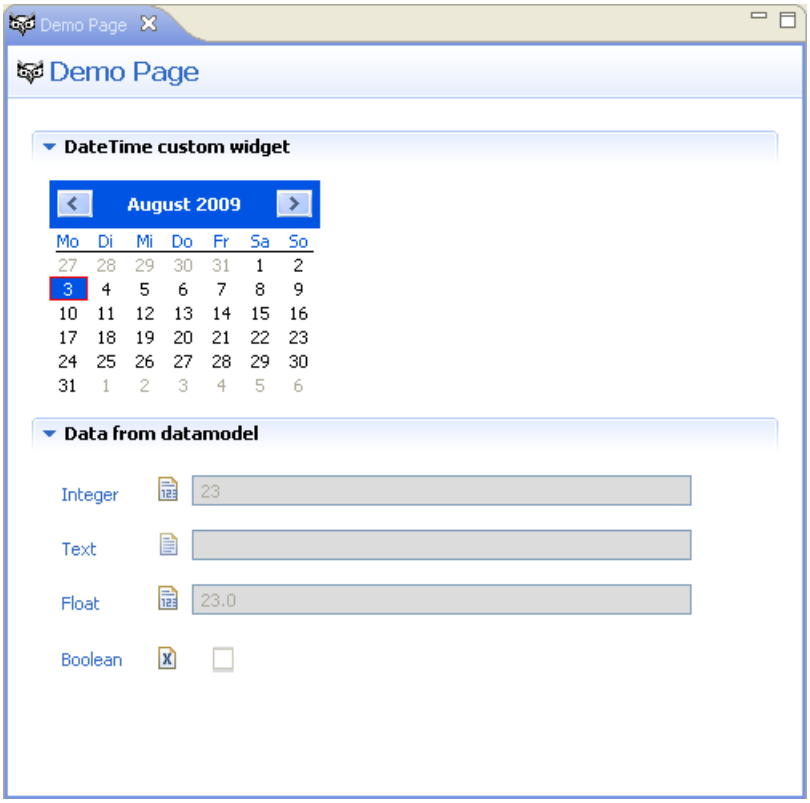


Figure 5.97. Demo2: Stand-alone editor with custom widget

From this demo you learn

- how to create and register your own custom widget.
- how to use PushServices to load data from the DataModel.
- how to update the enablement of your widgets while running the page.

5.21.6.3.2. Restricting the visibility

The Trigger for Demo2 is available in the *Sidebar* view only if a loaded project is selected. This loaded project must have target `TEST` and a module with module ID `PublicApiTest_TS_T00D0M0I0R0`. The visibility restriction is done in the `plugin.xml`registering the `dreisoft.tresos.guidedconfig.api.plugin.trigger` extension point.

If you disable or remove the module, the Trigger disappears from the *Sidebar* view.

5.21.6.3.3. Creating a custom widget

If the wizard is opened for the first time, the custom datetime widget shows the current day. You can change the value, which is then stored persistently. Thus after reopening the wizard, the last date is displayed again.

The values for day/month/year are attributes of a child Memento of the root Memento of the wizard. But changing the values for day/month/year is done in an atomic transaction. Since the wizard is a stand-alone editor, users may use the **undo/redo** buttons to undo or redo a selection. Then all data is changed with one click.

To add your own custom widget to your page, please refer to Section 5.21.7.6: Implementing a custom widget.

5.21.6.3.4. Using the PushService

The section *Data from datamodel* displays some values from the DataModel configuration. These values are loaded via a PushOperation via DCtxt API from the DataModel and stored in the Memento tree. As the demo does not use a PushOperation to store values back into the data model configuration, the parameters of this group are *read only*.

If you want to use the PushService, refer to Section 5.21.7.10: Using the PushService.

5.21.6.3.5. Updating widget enablement

The parameters of the section *Data from datamodel* are disabled, as they are not proposed to be changed. They are displayed only to show the values of the DataModel configuration. Changing the enablement of the widgets can be implemented in the `doUpdateWidgetEnablement` method of your `Backend` class.

For more information about the lifecycle of the widgets, have a look in chapter Section 5.21.4.5.6: Widget lifecycle.

5.21.6.4. Demo3

5.21.6.4.1. Overview

The demo3 provides an autoconfiguration wizard, which is available via the autoconfiguration dialog. Use the gear icon in the toolbar and open the *Autoconfiguration wizards ...* dialog.

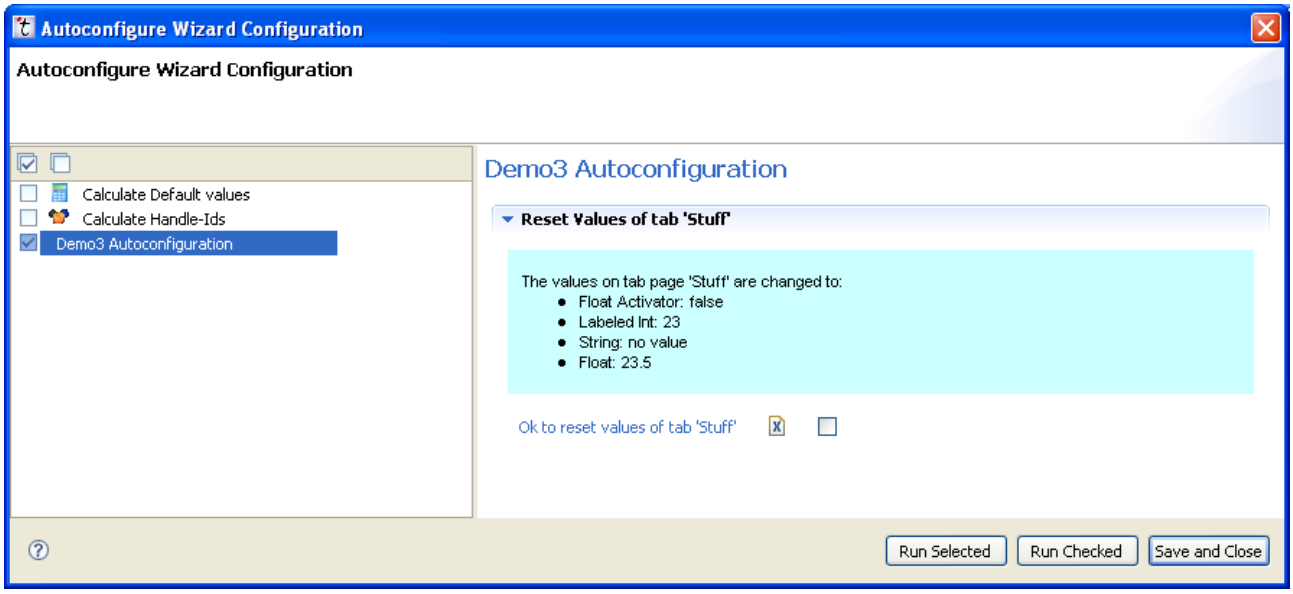


Figure 5.98. Demo3: Autoconfiguration dialog

In the left part of the autoconfiguration dialog you see the list of available wizards for the currently selected project. Each enabled wizard is displayed in the list below the gear icon with a *. Using the and buttons, users can check or uncheck all wizards with one click.

The right part of the autoconfiguration dialog is the configuration area for one wizard. This is updated if the selection in the left part has changed. If the wizard contains multiple pages, the pages are displayed on different tabs inside the right part of the autoconfiguration dialog.

In this demo you will learn

- how to use PushServices to change data of the DataModel
- how to create your own `ResultWidget` to display the result of the PushOperation.

Running this demo will help you to understand what can be done with PushOperations. Use this demo to reset the values of the DataModel configuration to predefined values. You find the parameters in the configuration tab page *Stuff*. The parameters are reset to the following values:

- *Float Activator* to false
- *Labeled Int* to 23
- *String* is made empty
- *Float* to 23.5

So to try this demo, open the generic editor for `PublicApiTest_TS_T00D0M0I0R0` and change the values of the parameters mentioned above of tab page 'Stuff' and save the configuration. Remember your values. Then open the autoconfiguration dialog 'Demo3' and enable the checkbox. Then you have several possibilities to start this wizard.

- Either directly select the wizard in the autoconfiguration dialog and press 'Run Selected'
- or check the wizard ☑ on the left side of the dialog and press 'Run Checked'
- or check the wizard ☑ on the left side and make sure it is the only wizard that is checked if there are more available and press **Save and close**. Then press the gear icon in the toolbar. Now all checked wizards of the autoconfiguration dialog are run.
- or click on the small arrow beneath the gear icon and select the *Demo3 Autoconfiguration* wizard from the list.

After running the wizard a result dialog appears with a list, which configuration data has been changed. You can open the generic editor again and you can see that the values have changed.

5.21.6.4.2. Using the PushService

If you run the wizard, a PushEvent is sent. All registered PushOperations are checked if they are listening to this event. All PushOperations for this event are run. In this case the PushOperation sets the four values and returns the paths to these configuration parameters to the PushEvent.

For more information about the PushService refer to Section 5.21.7.10: Using the PushService.

5.21.6.4.3. Registering a custom ResultWidget

The custom result widget is displayed on the second tab page inside the result dialog. It displays a table with two columns. The first column shows the path for the changed configuration parameter and the second column shows the new value of this parameter.

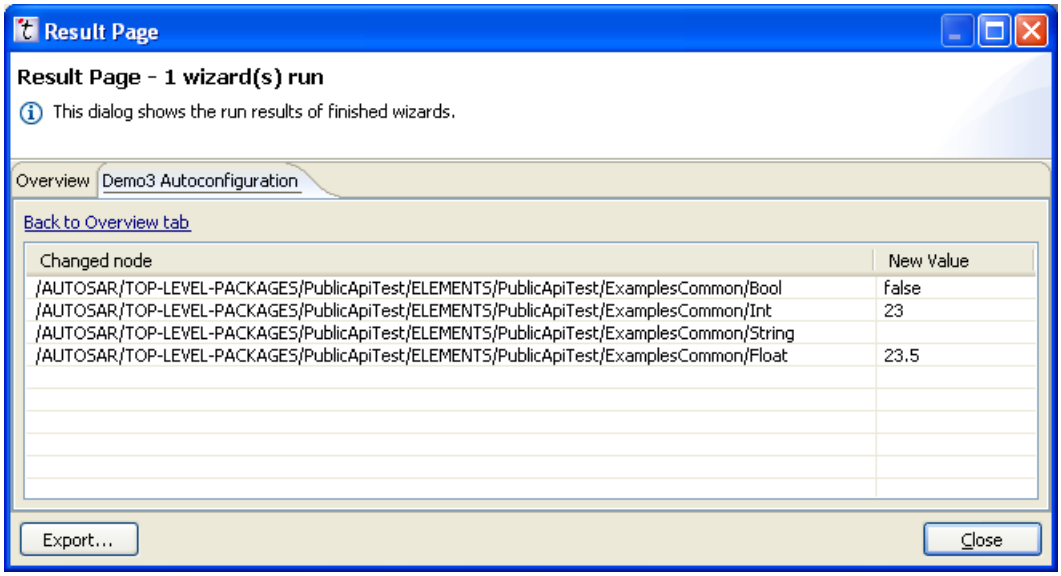


Figure 5.99. Demo3: Custom ResultWidget

Find more information on how to register an own ResultWidget in Section 5.21.7.11: Registering a custom ResultWidget

5.21.6.5. Demo4

5.21.6.5.1. Overview

The demo4 shows a custom module editor which is only available via *Project Explorer*.

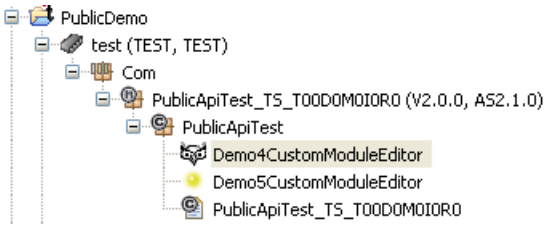


Figure 5.100. Demo4: Custom module editor in the *Project Explorer*

The demo contains examples of :

- all predefined widgets,
- how to validate parameters (text boxes, Table and TreeTable),
- selection events for buttons,
- filtering the rows of the Table,

- changing the sort column of the Table.

5.21.6.5.2. Restricting the visibility

For custom module editors it is not possible to restrict the visibility via extension point . This docking point is always available if the corresponding module is added to the project.

5.21.6.5.3. Displaying predefined widgets

This demo shows an example of all predefined widgets, that can be created via `WidgetFactory` class. A list which widgets are available via `WidgetFactory` can be found in Section 5.21.4.5.5: Predefined widgets

5.21.6.5.4. Validating

The following parameters display an error, if they are empty:

- Integer
- Float

For the Table and TreeTable the Boolean column is validated and a warning is displayed if the checkbox is not checked.

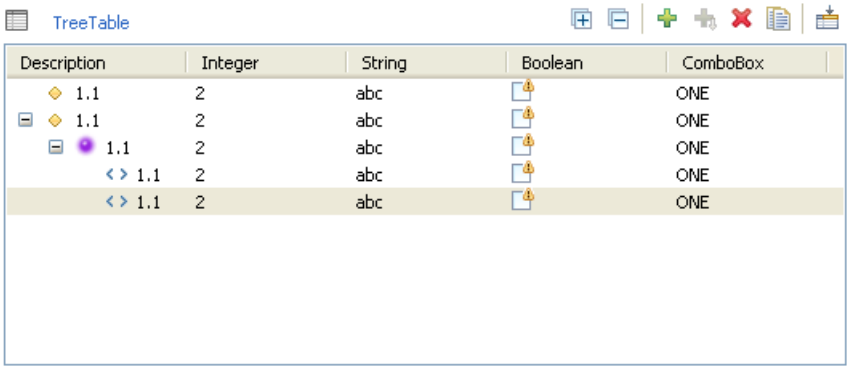


Figure 5.101. Demo4: TreeTable displaying errors

An example for validating the data you can find in Section 5.21.7.7: Validating the data.

5.21.6.5.5. Using selection events

This demo provides an example for selection events using buttons. There exist two buttons. One toggle button and one push button. The Table is sorted by the index column by default.

Table

I.	Integer	Text	Boolean	Combo	Float
0	47	hello		ONE	7.89
1	15	abc	☑	TWO	1.1
2	77	abc		ONE	3.27
3	2	hello		THREE	1.1

Figure 5.102. Demo4: Table before using the buttons

The button **ChangeSortColumn** changes the sorted column and the Table is sorted by the *Text* column.

Table

Index	Integer	Text	Boolean	Combo	Float
1	15	abc	☑	TWO	1.1
2	77	abc		ONE	3.27
0	47	hello		ONE	7.89
3	2	hello		THREE	1.1

Figure 5.103. Demo4: Table with changed sort column

The toggle button **Filter Table** filters all rows with value *hello* in column *Text* of the Table. To reset the row filter, press the toggle button again. Then all rows are displayed again.

Table

Index	Integer	Text	Boolean	Combo	Float
1	15	abc	☑	TWO	1.1
2	77	abc		ONE	3.27

Figure 5.104. Demo4: Table with row filter

If you want to know how to implement a selection event listener, please refer to Section 5.21.7.8: Using selection events.

5.21.6.6. Demo5

5.21.6.6.1. Overview

The wizard of demo5 is docked into multiple docking points of the GUI:

- as custom module editor in the *Project Explorer*, that opens a multi-page editor in the editor part of EB tresos Studio.

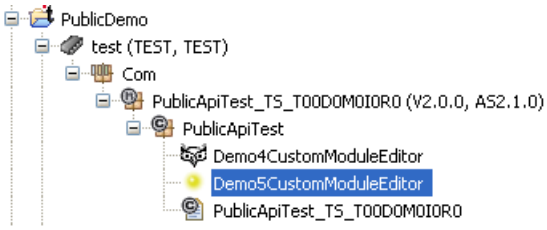


Figure 5.105. Demo5: Custom module editor in the *Project Explorer*

  As you can see a custom module editor can define a specific icon in the *Project Explorer* as known from the generic editor. The default icon is the owl.

- as dialog in the *Sidebar* view, where the multiple pages are navigateable via **back**/**next** buttons.

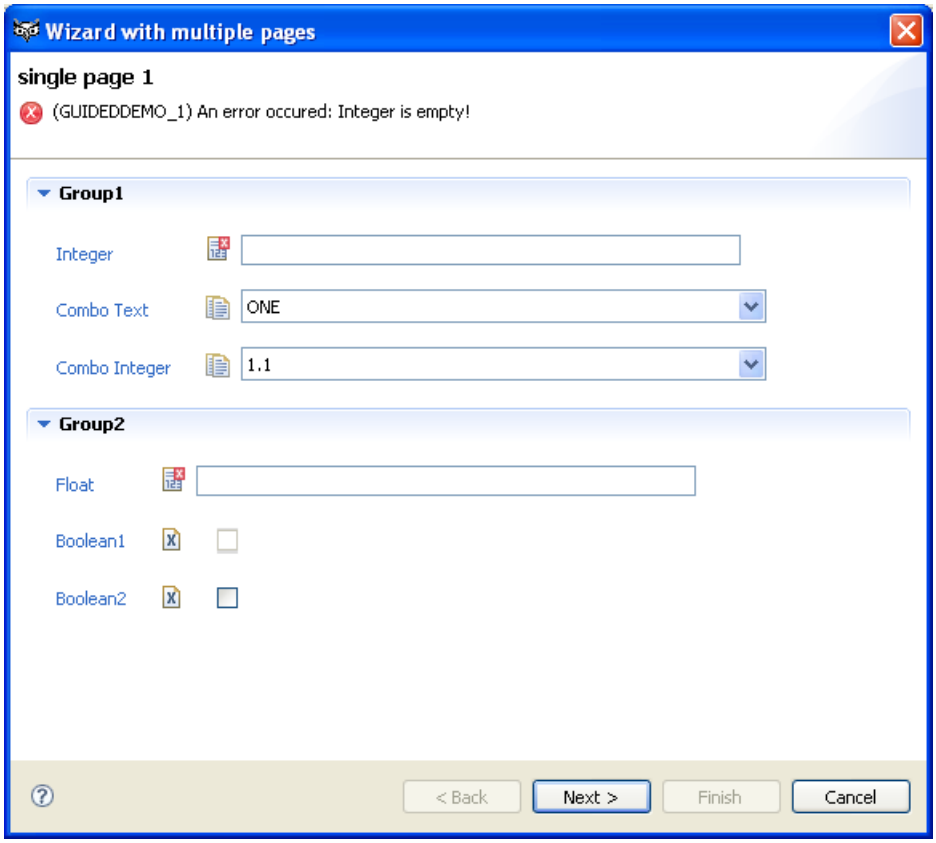


Figure 5.106. Demo5: Dialog available via *Sidebar* view

In this demo you will learn the differences between editors and dialogs:

- The navigation in multi-page wizards differs in editors and dialogs.

  In editors you navigate via the tab pages inside the editor part.

  Dialogs provide **back**/**next** buttons to navigate between several pages.
- Editors provide undo/redo support, dialogs do not.
- In editors all validation results are displayed as hyperlink on top of the page. It is possible to navigate to the errornous parameter via the error message.

  Dialogs display only one error at a time on top of the dialog. It is not possible to navigate to the errornous parameter.

5.21.6.6.2. Using multi-page support

A multi-page is registered as GUI page in the wizard extension point . The Multi-page class manages the single pages with the following methods

`doGetFirstPage()`
: returns the first single page

`doGetNextPage()`
: depending on the current page, the next page is returned

`doHasNextPage()`
: this manages the **Next** button in the dialog. You do not need to manage the way back through the pages, as the Guided Configuration API does this by remembering the page history.

5.21.6.6.3. Validating

When you open the dialog via the *Sidebar* view, a single error or warning is displayed at a time on top of the page. Warnings are only displayed if no errors exist anymore. Validation checks always the complete Memento, that contains all parameters of all pages. Thus the first page displays an error from the last page and vice versa. It is not possible to navigate to the errornous widget.

The **Finish** button is only enabled, if no errors exist anymore. You can finish the wizard, even if warnings are still contained.

If you use the custom module editor, all errors and warnings are displayed as summary on top of the page. Click on the hyperlink to open a dialog, in which you may select the errors and warnings. This causes the corresponding widget to get the focus no matter if it is available on the current page or on another.

For more information about validation, please refer to Section 5.21.7.7: Validating the data.

5.21.6.7. Demo6

5.21.6.7.1. Overview

The wizard for demo6 is available in the *Sidebar* view. In this demo you learn

- how to restrict the visibility of a wizard via the Backend implementation
- the usage of a special kind of label for the *Sidebar* view.

5.21.6.7.2. Restricting the visibility

The `Backend` class overwrites the method `doIsAvailable` and returns an `APIOperationStatus` if the current project does not provide system model support:

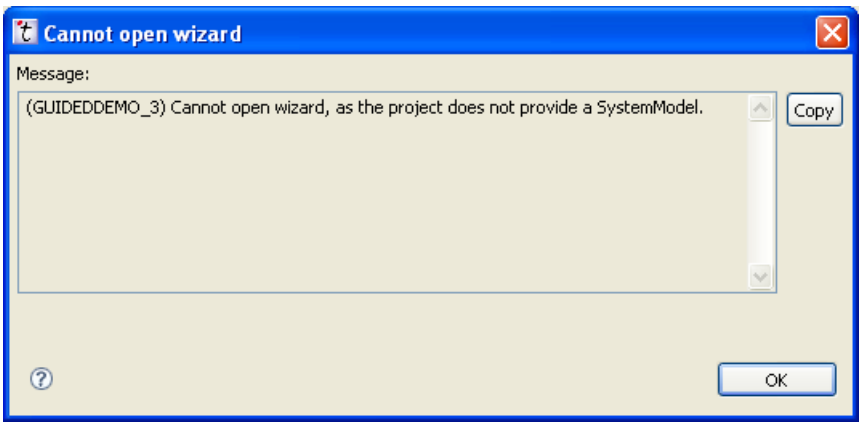


Figure 5.107. Demo6: Wizard cannot be opened

To be able to open the dialog, create a project with system model support and add a module with type `PublicApiTest`. Then you can open the wizard and no error dialog is displayed:

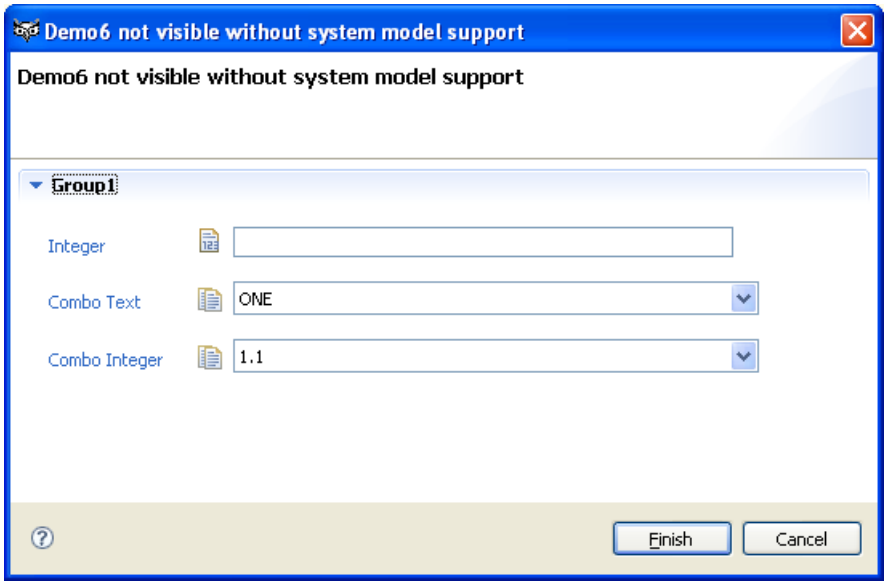


Figure 5.108. Dem6: Dialog if project provides system model support

5.21.6.7.3. Using hierarchical Sidebar labels

This demo uses a specific label inside the *Sidebar* view. The docking point is rendered as a tree.

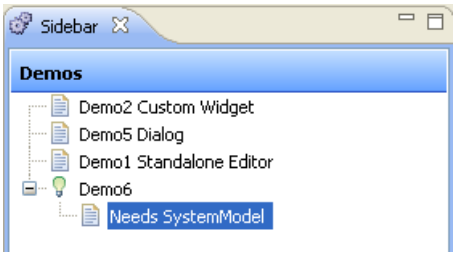


Figure 5.109. Demo6: Specific tree label in the *Sidebar* view

Define this label in the wizard registration in the extension point. The label can contain several `/`, each of which creates a new subnode in the tree. If there are several wizards registered that use the same subpath, they are all leaf nodes of the same tree branch.

```
<extension
      point="dreisoft.tresos.guidedconfig.api.plugin.trigger">
   <trigger>
      <sidebar
            categoryId="Demos"
            id="guidedDemo6"
            wizardId="guidedDemo6"
            wizardType="dialog">
         <visibility>
               <with  variable="ECUConfigContext.moduleType.PublicApiTest">
                  <equals  value="true"/>
               </with>
         </visibility>
         <display
               label="Demo6/Needs SystemModel"
               tooltip="Demo6 needs SystemModel">
         </display>
      </sidebar>
</trigger>
 </extension>
```

5.21.7. Common use cases

5.21.7.1. Registering a wizard

In this chapter you'll learn how to register and define a wizard.

Select the tab *Extensions* in the *Plug-in Manifest Editor*. Add the following extension point.

- `dreisoft.tresos.guidedconfig.api.plugin.guidedconfigwizard`

The extension point description can be found as a link in the plugin.xml editor.

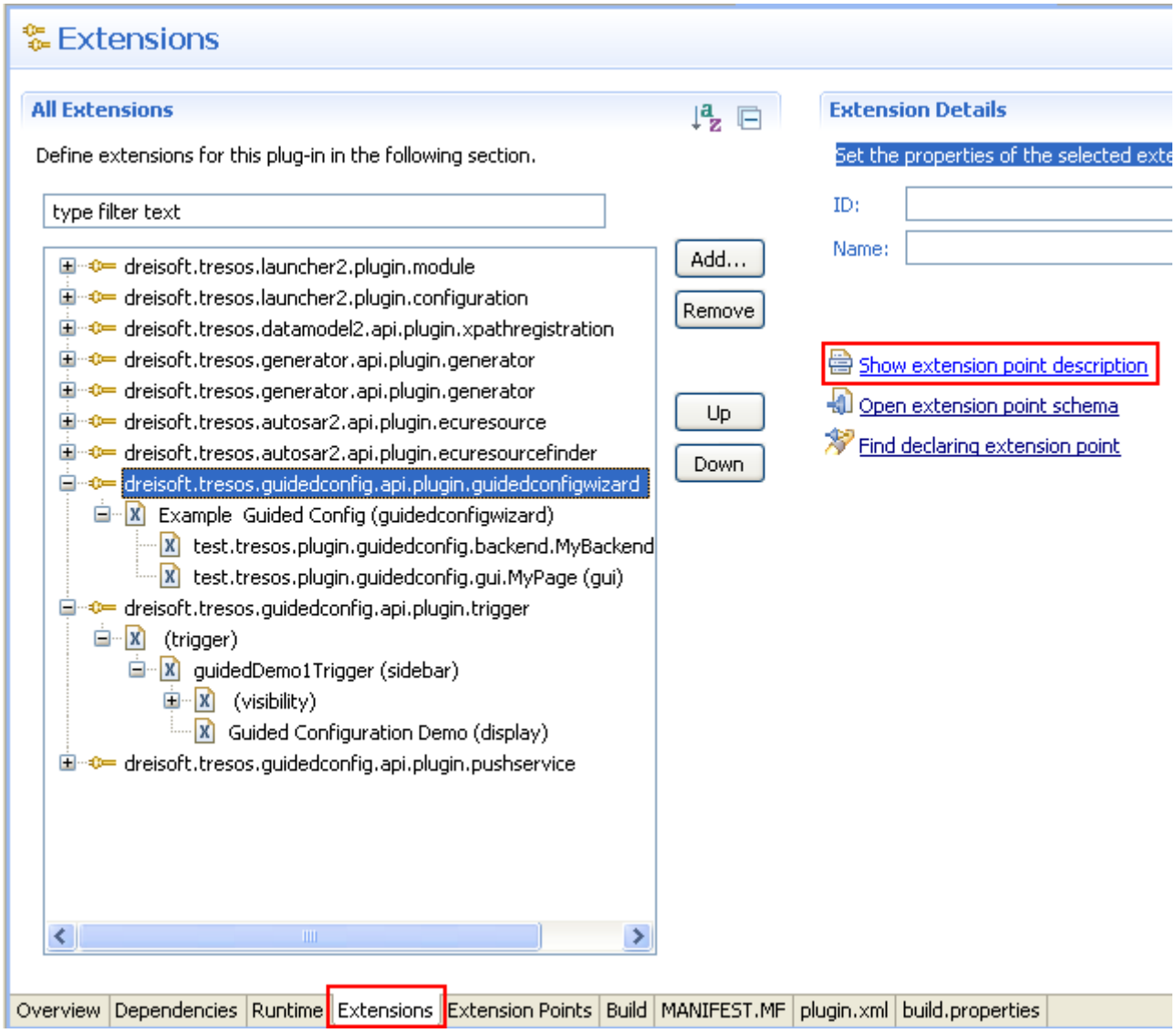


Figure 5.110. *Plug-in Manifest Editor*: Extensions tab

You can add several parameters, some of them are optional. But you must give the guided configuration wizard an ID. The ID will be referenced from the corresponding docking point. It is possible that multiple docking points share the same wizard registration. The following parameters are optional:

`icon`
: The path to an icon, that is displayed in the GUI. If you leave it blank, then the default icon is displayed.

`helpContextId`
: The helpContextId is a unique identifier that indicates the help context for this action.

`basePath`
: The path to the resources that are necessary for the wizard, e.g. subfolder `icons` for the icons for the GUI widgets.

`basePathPlugin`
: The plug-in if the basePath is not located in the same plug-in.

Now add the sub parameters:

`backend`
: The Backend class of the wizard.

`gui`
: The GUI representation of the wizard, either single or multi-page.

`resultGui`
: The ResultWidget is optional. If you do not add an own ResultWidget, then the `DefaultResultWidget` is used.

All three parameters (backend, gui and resultGui) have two entries themselves, the *class* and the *plug-in* entry. Whereas only the "class"-entry is of interest.

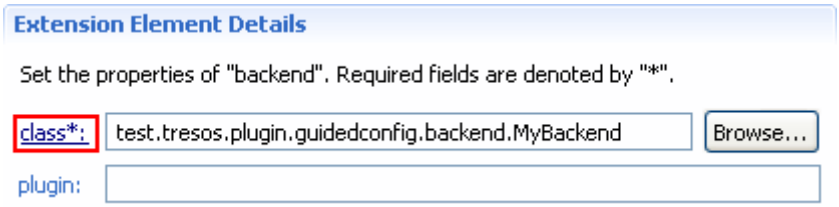


Figure 5.111. Create the classes

When you hit the link *class**, then the class creation dialog opens.The appropriate abstract classes to extend are already added to the class-creation-dialog. Just give the new class a name and a location and hit **Finish**.

5.21.7.2. Registering the docking point

This chapter provides an explanation on how to register the different kinds of docking points in the GUI. Find information about the docking points in Section 5.21.4.1: Wizard type overview

5.21.7.2.1. Sidebar editor and dialog

To add a stand-alone editor or dialog to the *Sidebar* view add the following extension point to your `plugin.xml`:

- `dreisoft.tresos.guidedconfig.api.plugin.trigger`

To expand the Trigger extension point, right-click on *(trigger)*, select *New*, and then choose *sidebar*.

- Assign a Trigger *ID* and enter the *wizard-ID* of your guided configuration wizard. Select either *editor* or *dialog* as *wizardType* from the drop-down menu.
- The field *categoryID* is for grouping several wizards. If more then one wizard has the same categoryID, they are displayed in the same group in the *Sidebar* view.
- Within the *display* element enter a label, that is displayed in the GUI, and a tooltip if you want. If the label contains slashes ("/"), the Trigger is displayed in a tree-element.
- The *visibility* element is used to add constraints for the visibility of the Trigger. Therefore information provided by the *Selection Context* is used. How to do this just have a look at the extension point description or refer to Section 5.21.7.3.1: Via extension point.

Create a new Trigger and choose *sidebar* trigger.

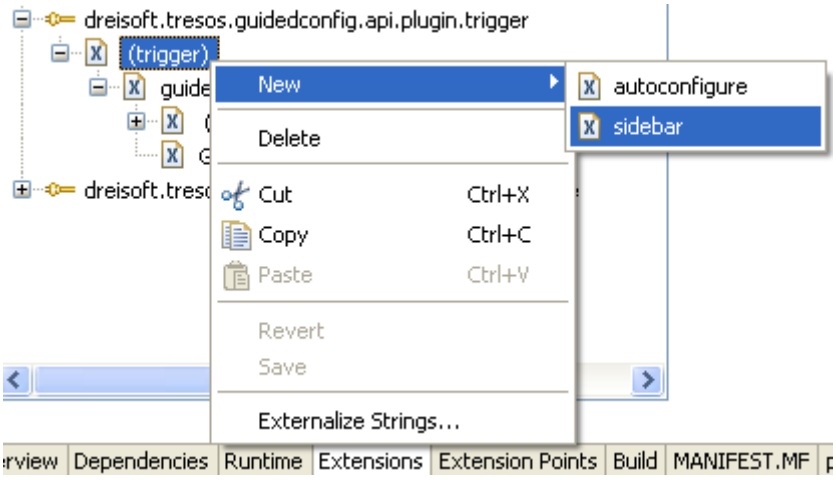


Figure 5.112. Add a sidebar trigger

Example of the registration for a stand-alone editor:

```
<extension
   point="dreisoft.tresos.guidedconfig.api.plugin.trigger">
   <trigger>
     <sidebar
         categoryId="Project"
         id="guidedDemoEditorTrigger"
         wizardId="guidedDemo1"
         wizardType="editor">
       <visibility>
            <with variable="ECUConfigContext.projectname"><equals value="test"/></with>
       </visibility>
       <display
            label="Guided Configuration Demo Editor"
            tooltip="Guided Configuration Demo Editor">
       </display>
     </sidebar>
   </trigger>
</extension>
```

The Trigger is displayed in the *Sidebar* view with the given label if the currently selected project is named *test*:

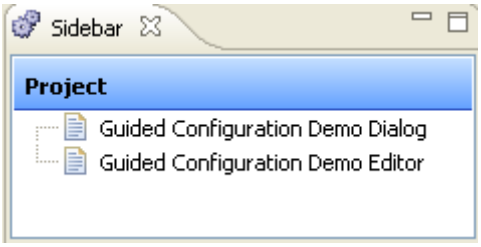


Figure 5.113. *Sidebar* view

Note

The *Sidebar* view is not displayed by default. It can be opened via Window -> Show View -> Wizard Sidebar.

You can register a dialog wizard the same way as the stand-alone editor. Give it a new label and ID. You need to change just the parameter `wizardType` in the extension point:

- `wizardType="dialog"`

5.21.7.2.2. Custom module editor

If you want to create a custom module editor, you need to register the following extension points:

- `dreisoft.tresos.launcher2.plugin.configuration`
- `dreisoft.tresos.launcher2.plugin.module`

and pass `dreisoft.tresos.guidedconfig.api.gui.editor.GuidedConfigEditor` as editor class, as well as a parameter *wizardId* with the ID of your wizard as value.

Note

A Custom Module Editor does *not* use the trigger extension point. It is registered via the module extension point.

Example for the registration of a custom module editor:

```
<extension point="dreisoft.tresos.launcher2.plugin.configuration"
     id="ModulePlugInConfiguration"
     name="ModulePlugIn Parameter Definition Extension">
     <configuration moduleId="ModulePlugIn_GuidedConfigdraft">
        <editor id="ModulePlugIn_GuidedConfigdraftEditor"
            label="GuidedConfigdraftEditor"
            tooltip="sample module">
              <class
              class="dreisoft.tresos.guidedconfig.api.gui.editor.GuidedConfigEditor"
                  plugin="dreisoft.tresos.guidedconfig.api.plugin">
                 <parameter name="wizardId" value="guidedDemo"/>
             </class>
        </editor>
      /configuration>
</extension>
```

5.21.7.2.3. Autoconfiguration wizard

To register an autoconfiguration wizard, add the following extension point to your `plugin.xml`:

- `dreisoft.tresos.guidedconfig.api.plugin.trigger`

To expand the Trigger extension point, right-click on *(trigger)*, select *New*, and then choose *autoconfigure*. Assign a Trigger ID and enter the ID of your guided configuration wizard.

Example for the registration of an autoconfiguration wizard:

```
<extension
   point="dreisoft.tresos.guidedconfig.api.plugin.trigger">
   <trigger>
      <autoconfigure
            id="guidedDemoAutoTrigger"
            wizardId="guidedDemoAuto">
         <visibility>
              <with variable="ECUConfigContext.projectname"><equals value="test"/></with>
         </visibility>
         <display
               label="Guided Configuration Demo Autoconfigure"
               tooltip="Guided Configuration Demo Autoconfigure">
         </display>
      </autoconfigure>
   </trigger>
</extension>
```

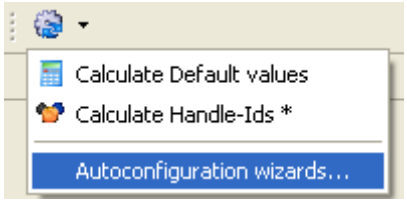


Figure 5.114. Autoconfiguration Menu

The autoconfiguration Trigger has more than one run option.

- You can click on the arrow and choose "Autoconfiguration wizards ..." which will open the autoconfiguration dialog. Inside this dialog all runnable autoconfiguration wizards can be configured. You can enable a checkbox for each trigger. This will enable a wizard to be run, when clicking on the gear icon in the toolbar of EB tresos Studio
- If you click on the gear icon, all wizards, that are checked in the autoconfiguration dialog, will be run.
- You can click on the arrow to open the dropdown list. Triggers with a "*" are checked inside the autoconfiguration dialog. But you can run each of them by just clicking on the Trigger entry in the list.

5.21.7.3. Restricting the visibility of a wizard

The visibility can only be restricted for Sidebar wizards (dialogs and stand-alone editors) and autoconfiguration wizards. The custom module editor is always available if the corresponding module is added to the project.

There are two ways to restrict the visibility of the wizard:

1. Section 5.21.7.3.1: Via extension point.
2. Section 5.21.7.3.2: Via Backend implementation.

5.21.7.3.1. Via extension point

To restrict the visibility via extension point, use the parameter `visibility` in the `dreisoft.tresos.guidedconfig.api.plugin.trigger` extension point. You may add several conditions in combination with `AND/OR` operations.

You can find the parameters which can be queried in chapter Section 5.21.4.2: Selection Context.

```
<trigger>
  <sidebar categoryId="Demos"  id="Demo1" wizardId="Demo1" wizardType="editor">
     <visibility>
        <and><with  variable="ECUConfigContext.target">
              <equals  value="TEST"/></with>
           <with  variable="ECUConfigContext.moduleType.PublicApiTest">
              <equals  value="true"/></with>
        </and>
     </visibility>
     <display label="Demo1Label" tooltip="Demo1Tooltip"/>
  </sidebar>
</trigger>
```

5.21.7.3.2. Via Backend implementation

The Backend provides a method `doIsAvailable`. To restrict the visibility you have to overwrite this method.

- If you don't want the Backend to further restrict the visibility, this method must return null.
- If you want to restrict the visibility, an `APIOperationStatus` containing the error message has to be returned.

The default implementation returns null.

If the visibility conditions are not met and an `APIOperationStatus` is returned, the message is displayed in a dialog. The dialog provides the possibility to copy the message from the dialog to the clipboard.

```
public APIOperationStatus doIsAvailable() {
   if( (getContexts().get( SystemConfigContext.ID ) == null)
   || (!getContexts().get( SystemConfigContext.ID ).isEnabled()) ) {
      return MyGuidedConfigOperationStatus.ERROR_OPEN_WIZARD();
   }
   return null;
}
```

5.21.7.4. Using XForms as GUI description

Instead of writing the GUI representation in Java, you can also register an XML file that defines the GUI.

Example to register an XForms file:

```
<extension
         point="dreisoft.tresos.guidedconfig.api.plugin.guidedconfigwizard">
      <guidedconfigwizard id="guidedDemo1">
         <gui class="dreisoft.tresos.guidedconfig.api.gui.xform.XFormMultiPage">
                 <parameter name="pageDescription" value="pages/Pages.xml"/>
         </gui>
         <backend class="test.tresos.plugin.guidedconfig.demo1.Demo1Backend" />
   </guidedconfigwizard>
 </extension>
```

You can find the widget descriptions in the `WidgetFactory` class. The Button provides the following parameters and their default values:

```
private final DefaultWidgetDescription DESC_BUTTON = new DefaultWidgetDescription(
    this,
    "button",
    new DefaultParameterDescription[]{
    new DefaultParameterDescription( "widgetId", String.class ),
    new DefaultParameterDescription( "mementoId", String.class, "__Button" ),
    new DefaultParameterDescription( "text", String.class, "" ),
    new DefaultParameterDescription( "icon", String.class, "" ),
    new DefaultParameterDescription( "flag", ButtonFlags.class, "PUSH" )} );
```

Example from the .xml file:

```
<Button widgetId="button" icon="icons/sample.gif" flag="TOGGLE"/>
```

A button is added with

- the widget-ID *button*
- an icon instead of text as label
- with type *TOGGLE*, which has two states ON/OFF.

In this example, not all parameters that are provided by the button widget are set. The other parameters are filled with their default values.

Some features for the GUI representation via Java code are not provided by the XForms description (e.g. adding selection events). But the Guided Configuration API provides the possibility to change the GUI representation right before the page is displayed.

To be able to change the GUI, register your own page class instead of `XFormMultiPage`. Your page class must extend the `XFormMultiPage`. Then overwrite the method `doAdaptWidget( ISWTWidget widget )`, to adapt the GUI representation of the given widget.

5.21.7.5. Creating the page

Instead of registering an XForms GUI description, you can write the GUI repesentation in Java.

You can choose to create:

- A single-page. Then your page class must extend `AbstractPage`.
- A multi-page.Then your page class must extend `AbstractMultiPage`.

An editor shows multi-pages in different tabs inside the editor. In a dialog the different pages can be reached with **Back/Next** buttons. This is managed by the Guided Configuration API.

Now it is time to fill the page with some widgets. Open your page class and add the following to the `doCreateControls(Composite parent)` method

```
// fetch the widget factory
WidgetFactory factory = getWidgetFactory();

// create a label
factory.createLabel( parent, "LABEL1", "LABEL_MEMENTO1", "Labeltext Integer" );
```

```
// create a textbox of type integer
factory.createText( parent, "TEXT1", "INT_MEMENTO1", TextFormat.INTEGER );

factory.layout( parent );
```

Note

`factory.layout( parent )` *MUST* be called at the end. If you like to change the layout afterwards feel free to do so, but it has to be called on the parent composite for at least one time, after all GUI elements were added.

The following methods need to be implemented:

- `doGetId()` has to return the page ID. Inside a multi-page this ID has to be unique.
- `doGetDescription()` has to return a description for the page.
- `doGetTitle()` has to return the page title.

5.21.7.6. Implementing a custom widget

A custom widget has to implement a controller and a view part.

The view part has to implement the interface `ISWTWidget`. The following methods have to be implemented:

`getWidgetId()`
: returns the widget-ID of the custom widget.

`getParent()`
: returns the parent composite of the custom widget

The controller part has to extend `AbstractWidgetController` to connect to the wizard lifecycle.

To connect the widget to a Memento, register it at the WidgetFactory by calling the following method:

```
registerWidget(ISWTWidget widget, String widgetId, String mementoId )
```

To provide undo/redo support, the operations on a Memento have to be done with the `MementoOperationHandler` methods. For more information about the `MementoOperationHandler`, please refer to Section 5.21.4.4.1: The MementoOperationHandler .

For an example how to add the custom widget to the page, refer to Section 5.21.6.3: Demo2

```
// create Custom Datetime widget
MyDatetime datetimeWidget = new MyDatetime( WIDGETID_DATETIME, customGroup );
// create controller for widget to receive events from the wizard engine
```

```
MyDatetimeController datetimeController = new MyDatetimeController(
                                                    datetimeWidget, MEMENTOID_DATETIME );
// register controller at the WidgetFactory
factory.registerWidgetController( datetimeController );
```

5.21.7.7. Validating the data

The guided configuration framework provides the possibility to validate the data of the widgets and show error and warning markers at the widget and on top of the page.

For using this functionality, you have to override the method `doValidate()` of your `Backend` class.

Example:

```
 public List<ValidationResult> doValidate() {
    // Initialize list of validation results
    ArrayList<ValidationResult> result = new ArrayList<ValidationResult>();

    // Get rootMemento of wizard
    Memento rootMemento = getMemento();

    // show an error if the integer field is empty
    String value = rootMemento.getString( Demo1MementoFactory.MEMENTOID_PAGE2_INTEGER );
    if( value == null || "".equals( value ) ) {
       result.add( new ValidationResult(
             Demo1MementoFactory.MEMENTOID_PAGE2_INTEGER,
             MyGuidedConfigOperationStatus.DRAFT_ERROR(
             "Integer field on page2 must not be empty!" ) ) );
   }
   return result;
 }
```

As the `ValidationResult` contains the Memento-ID and the message, the guided configuration framework can link from the message to the corresponding widget.

5.21.7.8. Using selection events

Some widgets provide events, so you can react on actions of the user. The following widgets provide selection events:

- Button
- Table
- TreeTable

Example for implementing a button selection event

```
protected void doCreateControls( Composite parent ) {
    // retrieve the widget factory
    WidgetFactory factory = getWidgetFactory();
    // add some widgets
       ...
    // create push button
    SWTButton sortButton = factory.createButton( group3,
                                                 "ChangeSortColumn",
                                                 "ChangeSortColumn",
                                                 "ChangeSortColumn",
                                                 "spy.gif",
                                                  ButtonFlags.PUSH );
   // create your own ISelectionEventListener
   SortColumnSelection sorting = new SortColumnSelection();
   // add SelectionListener to the button widget
   sortButton.getController().addSelectionEventListener( sorting );
   // change tooltip text
   sortButton.setToolTipText( "Change sort column to column 'Text'." );
   factory.layout( parent );
}

 private static class SortColumnSelection implements ISelectionEventListener {
       public void selectionChanged( WidgetSelectionEvent event ) {
                  if( !(event instanceof ButtonSelectionEvent) ) return;
                   m_table.setSortColumn( MyEclipseNLS.LABEL_TEXT(),
                                          SortFlags.COLUMN_ORDER_ASCENDING );
         }
 }
```

5.21.7.9. Creating a MementoFactory

Per default the `DefaultMementoFactory` is used and all Mementos are persistent. You are able to change this behaviour by creating an own `MementoFactory`.

A custom MementoFactory must extend `AbstractMementoFactory` and implement only one method `getMemento( Memento parent, String name )`. This method can be used to

- set default values or calculate values for the Mementos
- decide whether a Memento is to be persistent or not.

```
public class MyMementoFactory extends AbstractMementoFactory {
  public static final MyMementoFactory INSTANCE = new MyMementoFactory();
  /*
  * Create a new Memento with the given name and
  * set default values for the root memento
  */
  public Memento getMemento( Memento parent, String name ) {

       Memento memento = new Memento( name, this );
       if( parent == null ) {
         memento.setString( IConstants.MEMENTO_KEY_TEXT1, "defaulttext1" );
         memento.setString( IConstants.MEMENTO_KEY_TEXT2, "defaulttext2" );
       }
       return memento;
  }
}
```

5.21.7.10. Using the PushService

In this chapter you will learn how to register a PushOperation and how to start this operation by sending a PushEvent.

5.21.7.10.1. Registering a PushOperation

Select the tab *Extensions* in the *Plug-in Manifest Editor*. Add the following extension point.

- `dreisoft.tresos.guidedconfig.api.plugin.pushservice`

After adding the extension point, assign an ID, a name and a description and create the PushOperation class by clicking on the *class* link.

Via the Event tag the PushOperation can be restricted to a special event or *Selection Context*.

```
<extension point="dreisoft.tresos.guidedconfig.api.plugin.pushservice">
 <pushoperation
       desc="Example PushOperation"
       id="DemoPushOperation"
       name="Example Push Operation">
    <operationclass
           class="test.tresos.plugin.guidedconfig.pushservice.operation.ExamplePushOperat
    <event>
        <with variable="class">
            <equals value="test.tresos.plugin.guidedconfig.pushservice.operation.ExampleE
        </with>
    </event>
 </pushoperation>
</extension>
```

5.21.7.10.2. Sending a PushEvent

The event has to extend the `AbstractPushEvent` class.

```
// Create a PushEvent that is sent to a PushOperation via the PushService
ExampleEvent pushEvent = new ExampleEvent( ExampleEvent.EVENT_TYPE_AUTOCONFIGURE );
pushEvent.setBackend( this );
pushEvent.setSourceMemento( rootMemento );
pushEvent.setTargetContext( getTargetConfigurationContext( MODULE_SCHEMA_PATH ) );

// Trigger the PushOperations
PushService.getInstance().callSync( pushEvent, false );

// Get the list of changed nodes
List<DCtxt> changedNodes = pushEvent.getChangedNodes();
```

5.21.7.11. Registering a custom ResultWidget

In this chapter you will learn how to register a custom ResultWidget.

The custom ResultWidget has to extend the `AbstractResultWidget` class. Your class then has to be registered in the `dreisoft.tresos.guidedconfig.api.plugin.guidedconfigwizard` extension point:

```
<extension
       point="dreisoft.tresos.guidedconfig.api.plugin.guidedconfigwizard">
    <guidedconfigwizard id="guidedDemo3">
       <backend class="test.tresos.plugin.guidedconfig.demo3.Demo3Backend"/>
       <gui class="test.tresos.plugin.guidedconfig.demo3.Demo3Page"/>
       <resultGui class="test.tresos.plugin.guidedconfig.demo3.Demo3ResultWidget"/>
    </guidedconfigwizard>
</extension>
```

5.22. Installing the Public Api Demo Plugin

Note

Precondition: Eclipse has to be setup as described here Section 5.18: Setting Up Eclipse

All Public Api examples in this DevelopersGuide can also be found in the demo plug-in *PublicApiTest_TS_T00D0M0I0R0* which is located in *<tresos-install-dir>/demos*. This demo plug-in can be imported to your eclipse installation by right-clicking in the Package Explorer and select *Import... -> Import Existing Project Into Workspace*

The next step is to generate NLS/OS (Section 5.19.3: Calling The NLS/OS-Tool). Now the plug-in is ready for use. It can also be exported, to be used directly within the tresos Studio installation. To do so just right-click the plug-in project and hit export. Select *Deployable plug-ins and fragments* and hit *next*. Check the project, select a folder to where it shall be exported and uncheck the option *package plug-ins as individual JAR archives*. That's it. Now copy the exported plugin to your tresos Studio plugins folder. After a restart of tresos Studio, a new project can be created with architecture/derivate TEST/TEST with version 2.1. Add the module PublicApiTest and finish.

6 Bibliography

Bibliography

[1] *Autosar Layered Software Architecture*. Version 2.1.0, Revision 0014. Publisher: Autosar.

[6] *OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection*. Author: Zimmermann. Pages: 425 to 432.

[11] *Autosar Model Persistence Rules for XML*. Version 2.1.2, Release 3.0, Revision 0001. Publisher: Autosar.

[18] *LIN Specification Package*. URL: http://www.lin-subbus.org/. Publisher: Lin Consortium.

Glossary

Adc	`Analog Digital Converter Drivermodule`: The `Adc Driver` initializes and controls the internal Analog Digital Converter unit of the microcontroller. The driver is equipped with a set of basic functionalities with single value result access mode. Enhanced functionalities with stream access are available, too. Typical applications in which this sort of specific analog conversion is requested are knocking detection or valve control position via spectrum analysis of the current on the coil. The `Adc Driver` is part of the microcontroller abstraction layer.
ADC	analog-to-digital converter
alive supervision	The alive supervision is a feature of the `WdgM`. The `Supervised Entities` which are parts of the SW-Cs trigger the `WdgM` according to a configured timing schema. If there occurs a timeout on such a trigger cycle, the alive supervision fails and the WdgM itself stops triggering the Watchdog hardware.
alive timeout	Alive timeout (or also known as communication timeout) is a feature, which can be enabled in the data receiver Com specification of a data element prototype with data semantics (`isQueued = FALSE`). In case of inter-ECU sender/receiver communication, the `Rte` then monitors the timely reception of the data, i.e. if no new value has been received within a certain time period, an error is reported to the application. For intra-ECU communication, time-out monitoring is not supported.
API	Application programming interface
application error	An application-specific error that is returned by a server to the client. Application errors are defined in software component description as a part of the client/server interface.
application header file	A software component's application header file contains the definition of the `Rte`'s API and associated data structures that are generated for the software component. One application header file is generated for each atomic software component type encountered in the input. Its name is `Rte_<appl>.h` whereas `<appl>` is the name of the atomic software component type. The application header files are both generated in the contract and in the generation phase and must be included by the software component's code.
argument prototype	Argument prototypes describe the arguments for an operation prototype in a client/server interface. The arguments are passed to the operation prototype in the sequence in which they are defined in the input. Each argument prototype has a type and a direction. The direction indicates if the argument prototype is an

- input parameter (`direction = IN`, read but not written by the operation prototype)
- input/output parameter (`direction = INOUT`, read and written by the operation prototype
- output parameter (`direction = OUT`, not read but written by the operation prototype)

asynchronous server call point	Property of a runnable entity which indicates that the runnable entity asynchronously invokes an operation prototype in the provide port of a client/server interface by using the appropriate Rte API call. Asynchronous invocation of an operation prototype implies that the Rte API call does not block until the execution of the server has terminated and that a separate Rte API call is necessary to retrieve the result.
asynchronous server call returns event	An asynchronous server call returns event is an *Rte event* that is issued by the `Rte` to notify the client in a client/server connection that either the server execution has terminated or a timeout or communication error has occurred. Only one asynchronous server call returns event is issued by the `Rte` for each asynchronous invocation of an operation prototype. An asynchronous server call returns event can trigger both a runnable entity and resolve a wait point.
atomic software component	An *atomic software component* encloses application software, realizing a particular functionality that cannot be divided any further. Therefore, atomic software components are the smallest software components that are executed on a particular ECU. The atomic software component type describes the communication interface (external view) of the atomic software component, i.e. the provide and require ports with their provided or required interfaces. Each atomic software component type must have exactly one internal behavior that specifies its functionality in the form of runnable entities and Rte events that trigger the runnable entities (internal view). An implementation has to be selected for the internal behavior as part of the Rte configuration process.
atomic software component instance	An *atomic software component instance* of an *atomic software component type* is created by specifying a component prototype of the atomic software component type within a composition type. If the internal behavior of the atomic software component type supports multiple instantiation, more than one component prototype of the atomic software component type can be created. Since the composition type can again be instantiated, an atomic software component instance does not necessarily correspond to a component prototype. For the generation of the `Rte`, only atomic software component instances are relevant. The system is flattened and reduced to atomic software component instances, all compositions are removed.
EB tresos AutoCore	Elektrobit's implementation of an Autosar basic software stack (`BSW`).

Autosar	*Aut*omotive *O*pen *S*ystem *AR*chitecture
Autosar types header file	Is a header file generated by the Rte generator that contains ► the definition of all data types and enumeration constants encountered in the input as well as ► the definition of all data types that are required by the Rte implementation ► except the software component specific type definitions. The latter are defined in the software component's application header file.
basic storage object	A *basic storage object* is a sub-structure which is used to administrate and to store non-volatile data. A basic storage object can reside in different memory locations, such as RAM, ROM, or NVRAM.
block management type	The (configurable) individual composition of an *NVRAM block* in chunks of mandatory/optional *basic storage objects* is called a *block management type*.
BSW	*B*asic *sofw*are; defined by Autosar the basic software includes the following software modules ► microcontroller abstraction driver ► ECU abstraction ► communication stack ► Autosar services ► operating system ► complex device drivers
calibration parameter	Calibration parameters are parameters which influence the ECU behavior. Examples for such parameters are variant-dependent data, run-time software switches, process controlling data, interpolation curves or interpolation fields. Calibration parameters are typically determined during an calibration process and then stored into ROM or non-volatile RAM. In Autosar, calibration parameters can be defined within one atomic software component or within a calibration software component. Calibration parameters defined for an atomic software component can only be used within the atomic software component. Several instances of the same atomic software component type can either share the same instance of a calibration parameter or use their own instances. The `Rte` provides the API `Rte_CData` to access the calibration parameters.
calibration software component	A calibration software component is a software component which provides calibration parameters to other software components. Calibration components are used when a calibration parameter is to be shared among different application components. A calibration software component provides ports with a

calibration interface. The calibration interface defines calibration parameters which are provided to other atomic software components. Ports of atomic software components can be connected to the calibration components via assembly connectors. The `Rte` provides the `Rte_Calprm` API to read the calibration parameters.

`Can`

`CAN Driver` module: The `CAN Driver` performs hardware access and offers a hardware independent API to the upper layer. Only the `CAN Interface` has access to the `CAN Driver`. It provides services to control the behavior and state of the CAN controllers that belong to the same CAN hardware unit. The `CAN Driver` furthermore offers the functionality for performing communication on the bus (e.g. sending messages). The `CAN Driver` is part of the microcontroller abstraction layer. It is the lowest layer of the CAN bundle.

`CanIf`

`CAN Interface` module: The `CAN Interface` is the hardware independent connection of the `CAN Driver (Can)` and the `CAN Transceiver Driver (CanTrcv)` to the upper layer modules which want to communicate on the CAN bus. It provides mode handling (including bus-sleep and wakeup if supported by `Can` and/or `CanTrcv`) on a per network base and offers a hardware independent interface for message communication that maps PDU IDs to the corresponding `CAN Driver` parameters and vice versa. It furthermore provides some extended functionalities, e.g. dynamic transmit CAN IDs, receive buffering and polling, which can be configured during configuration. The `CanIf` is part of the ECU abstraction layer.

`CanNm`

`CAN Network Managementmodule`: The `CAN Network Management` is responsible for the cyclic transmission of network management messages on the CAN bus. The `CAN Network Management` is part of the communication services inside the Autosar service layer. It is positioned below the `Generic Network Management Interface (Nm)`and uses the `CAN Interface (CanIf)` to send and receive network management messages.

`CanSm`

The `CAN State Management` module is the CAN specific state manager, handling the control flow of the associated CAN buses. The `CAN Network Management` is part of the communication services inside the Autosar service layer. The `CanSm` implements the CAN specific network mode state machine which handles the states full-, silent- and no- communication.

`CanTp`

`CAN Transport Layermodule`: The `CAN Transport Layer` is part of the service layer. The `CAN Transport Layer`is responsible for sending and receiving diagnostic messages. It furthermore segments and reassembles messages which do not fit to a single CAN frame.

`CanTrcv`	`CAN Transceiver Driver` module: The `CAN Transceiver Driver` is part of the communication hardware abstraction in the ECU abstraction layer.
client/server interface	A client/server interface aggregates all operation prototypes that are provided by a server and can be invoked by a client. A client/server interface must contain at least one operation prototype. A client/server interface may define possible application errors.
`CNm`	`CAN Generic Network Management` module: Obsolete module in Autosar 3.0 and up. Functionality of this module has been moved into `CanNm`.
code type	The code type attribute of an implementation specifies whether an atomic software component is delivered as a source code component (`code type = SRC`) or as an object-code component (`code type = OBJ`). Optimized access to operation prototypes is only available for atomic software components delivered as source code components.
`Com`	`Communication` module: The `Com` module implements the packing of signals into PDUs and initiates the transmission on sender side. On the receiver side the signals are unpacked from the PDUs, decoded and the applications are informed about the reception via the `Rte`. The `Com` module is located in the communication stack above the `PduR` and below the `Rte`.
Common Published Information	*Common Published Information* are read-only configuration parameters, such as version information, module-ID and vendor-ID.
`ComM`	`Communication Manager` module: The `Communication Manager` simplifies the resource management for the `EcuM`, which in turn simplifies the resource handing of the communication stack. The `Communication Manager` is located in the system services in the service layer.
compatibility mode	Compatibility mode of the Rte generator: The compatibility mode ensures compatibility of Rte generators by different vendors between contract phase and generation phase. Do not confuse with OSEK compatibility mode.
composition	Compositions enable a hierarchical software architecture of an Autosar system. The composition type describes the communication interface (external view) of a composition, i.e. the provide and require ports with their provided or required interfaces that are visible outside the composition. Furthermore, the composition type consists of component prototypes, which structure the composition's functionality, and of connectors, which either connect the ports of the composition's component prototypes or make them visible to the outside of the composition type. Unlike atomic software component types, no internal behavior or implementation is specified for a composition type. A system must have at least one composition which is the top level composition.

Com signal	Messages handled by Autosar `Com` for reception and transmission. They were called Com messages in OSEK Com.
component prototype	A component prototype is the instantiation of an atomic software component type or a composition type within a composition type. Since the composition type itself can also be instantiated again within another composition, a component prototype does not necessarily correspond to an atomic software component instance.
connector	Connectors are part of a composition type. An assembly connector connects one provide port of a software component instance with one require port of another software component instance within the composition type. A delegation connector makes a port of a software component instance within the composition type visible to the outside of the composition type. This is done by connecting a require port of the software component instance with a require port of the composition type or by connecting a provide port of the software component instance with a provide port of the composition type. A service connector prototype connects a port of a service component prototype to a port of an application software component instance. In contrast to assembly connectors, service connector prototypes can be used across composition boundaries, i.e. no delegation connectors have to be used.
contract phase	The first of two generation phases distinguished by an Rte generator. Only the application header files and the Autosar types header file are generated during the contract phase. With the application header file and the Autosar types header file it is possible to compile an atomic software component's code and deliver the software component as an object-code component. If different Rte generators are used for both generation phases, they have to operate in compatibility mode (see operating modes).
`Crc`	`CRC Routines` module: In the Autosar environment Crc routines are used by the `NVRAM Manager`, for example. The `Crc` is located in the system services in the service layer.
CRC	A cyclic redundancy check (CRC) is a type of function that takes an input of data stream of any length and produces a value of a certain fixed size as output. The term *CRC* is often used to denote either the function or the function's output. A CRC can be used as a checksum to detect alteration of data during transmission or storage.
data element prototype	Represents the data elements that can be sent/received in a port that provides/requires the sender/receiver interface. A data element prototype is aggregated by a sender/receiver interface. It has a type and either data (`isQueued =FALSE`; last-is-best) or event (`isQueued =TRUE`) semantics.
data element invalidation	A feature which can be enabled in the data sender com spec of a data element prototype with data semantics (`isQueued =FALSE`). On the sender side, the

	`Rte` provides an API to invalidate a data element prototype. On receiver side, the reception of an invalid value is either handled as an error or the data is silently replaced by the invalid value.
data read access	Property of a runnable entity that indicates that the runnable entity has access to the value of a data element in a require port that was received when the runnable entity was activated. That value does not change during the execution of the runnable entity. The `Rte` has to provide a copy of the data element and has to ensure that it does not change while the runnable entity is executed. If alive timeout or Invalidation is specified for the data element, the existence of data read access for the data element prototype in the require port also implies that the runnable entity has access to the status information that was returned to the `Rte` when the local copy of the data element prototype was updated.
data receive error event	A data receive error event is issued by the `Rte` to notify the receiver in a sender/receiver connection for a data element prototype with data semantics that either the available value is outdated or it has been invalidated. A data receive error event can only trigger a runnable entity but is not allowed to resolve a wait point.
data received event	A data received event is issued by the `Rte` to notify the receiver in a sender/receiver connection for a data element prototype that a new value is available. If the corresponding data element prototype has data semantics (`isQueued =FALSE`), the data received event is only allowed to trigger a runnable entity. If the corresponding data element prototype has event semantics, the data received event can both trigger a runnable entity and resolve a wait point.
data receive point	Property of a runnable entity which indicates that the runnable entity explicitly reads a data element prototype in a require port by invoking an Rte API call.
data receiver Com specification	A data receiver Com specification specifies communication attributes for a data element prototype with data semantics (`isQueued =FALSE`) that is received by a component in a require port. For a data element prototype with data semantics, the attributes `alive timeout` (optional) and `init value` (optional) can be specified. Furthermore, a data receiver Com specification allows to specify reception filters for a data element with data semantics.
data send completed event	A data send completed event is an Rte event that is associated with a data send point and that is issued by the `Rte` to notify the sender in a sender/receiver connection that the transmission was successful or that a timeout or communication error was detected. For transmission of a data element prototype, only one data send completed event is issued by the `Rte`. A data send completed event can both trigger a runnable entity and resolve a wait point.
data sender Com specification	A data sender Com specification specifies communication attributes for a data element prototype with data semantics (`isQueued =FALSE`) that is sent by

a component prototype in a provide port. For a data element prototype with data semantics, the attributes `canInvalidate` (optional), `init value` (optional) and `transmission acknowledgement` (optional) can be specified.

data send point

Property of a runnable entity which indicates that the runnable entity explicitly requests transmission of a data element prototype through a provide port by invoking an Rte API call. If `canInvalidate` or transmission acknowledgement is specified for the data element prototype in the provide port, the existence of a data send point also implies that the Rte API for invalidating the data element prototype or for collecting the transmission acknowledgement is available for the runnable entity.

data write access

Property of a runnable entity which indicates that the runnable entity provides a new value for transmission of a data element prototype through a provide port after the runnable entity has terminated. The `Rte` has to ensure transmission after the termination of the runnable entity. If `canInvalidate` is specified for the data element prototype in the provide port, the existence of data write access also implies that the invalid value can be set during execution of the runnable entity and is then transmitted after the runnable entity terminates.

dataset NV block

The dataset NV block is a configurable type of an NV block.

DBC

The DBC format is a legacy format used to describe CAN communication data. It is a proprietary format by Vector Informatik.

`Dbg`

Short form of the Autosar `Debugging` module.

`Dcm`

`Diagnostic Communication Manager` module: The `Diagnostic Communication Manager` manages diagnostic protocols (UDS, OBD) inside the ECU. It monitors

- protocol communication states,
- timings,
- life sign signals, etc.

and dispatches diagnostic requests to the responsible software components. The `Diagnostic Communication Manager` is located in the communication services in the service layer.

`Dem`

`Diagnostic Event Manager` module: The `Diagnostic Event Manager` processes and stores diagnostic events (errors) and associated environment conditions reported by the application layer or the Autosar BSW. It provides fault information to the `Diagnostic Communication Manager` (`Dcm`) for off-board diagnostics. The `Diagnostic Event Manager` is located in the communication services in the service layer.

`Det`	`Development Error Tracer` module: The `Development Error Tracer` (`Det`) is a debugging API to detect software bugs during the software development and integration phase. The `Development Error Tracer` is located in the system services in the service layer.
DID	Debugging IDentifier
`Dio`	`Digital IO Driver` module: The `Dio` module allows level read/write of single port pins (channel), a port pin group (channel group) or a complete port. The `Dio Driver` is part of the microcontroller abstraction layer.
direct (function) call	Optimization for synchronous server calls. For a direct function call, no context switch is performed, the server runnable entity is executed within the client's task context. This is realized by directly mapping the Rte API to the C function of the server runnable entity. Certain criteria have to be fulfilled so that a direct function call can be made.
`Ea`	`EEPROM Abstraction` module: The EEPROM Interface provides equal mechanism to access microcontroller internal and external EEPROM devices. It abstracts from the location of peripheral EEPROM devices (internal or external), the ECU hardware layout and the number of EEPROM devices. The EEPROM is part of the memory hardware abstraction in the abstraction layer.
EB tresos	The Elektrobit tresos product line
Eclipse	Eclipse in an open-source framework. Eclipse is a multi-language software development platform comprising an IDE (integrated development environment) and a plug-in system to extend it. It is written primarily in Java and is used to develop applications in this language and, by means of the various plug-ins, in other languages as well C/C++, Cobol, Python, Perl, PHP and others.
ECU	Electronic control unit
EcuC	The virtual Ecu configuration module is defined by Autosar to hold the global PDUs. To be able to define a PDU flowing through the COM stack two modules need to be able to refer to the same PDU object. Therefore a generic PDU container has been defined which does not belong to any module but to the whole COM stack. All COM stack modules dealing with PDUs have a reference to such a global PDU.
`EcuM`	`ECU State Manager` module: Basic software module that manages all aspects of the ECU related OFF, RUN and SLEEP states and the transitions between these states such as STARTUP and SHUTDOWN. Management of all wakeup events and configuration of the ECU for SLEEP when requested The `Ecu State Manager` is part of the system services in the service layer.
`Eep`	`EEPROM Driver` module: The `EEPROM Driver` provides services for reading, writing, erasing to/from an EEPROM. It also provides a service for

comparing a data block in the EEPROM with a data block in the memory. The `Eep` is part of the memory driver in the microcontroller abstraction layer.

entry point

The entry point (attribute `Symbol`) of a runnable entity specifies the name of the C function that implements the runnable entity. The names of the entry points of all runnable entities on one ECU have to be unique. The same entry point cannot be shared by two or more runnable entities.

EEPROM

*E*lectrically *e*rasable *p*rogrammable *r*ead *o*nly *m*emory; is a type of non-volatile memory used in computers and other electronic devices to store small amounts of data that must be saved when power is removed, e.g., calibration tables or device configuration. When larger amounts of static data are to be stored (such as in USB flash drives) a specific type of EEPROM such as flash memory is more economical than traditional EEPROM devices. EEPROMs are realized as arrays of floating-gate transistors.

exclusive area

A mechanism of the `Rte` to specify critical sections. A runnable entity can request access to an exclusive area to prevent preemption by other runnable entities that have access to the same exclusive area in order to ensure data consistency when accessing shared data. A runnable entity can either access an exclusive area implicitly (runs in exclusive area, i.e. the whole execution of the runnable entity is protected by the exclusive area) or explicitly (can enter exclusive area, i.e. Rte API calls are provided to enter/leave the exclusive area). For an exclusive area, different implementation mechanisms can be configured:

- enabling/disabling of interrupts
- usage of `Os` resources
- usage of non-preemptive tasks
- cooperative runnable placement

`Fee`

`Flash EEPROM Emulation` module: Emulates EEPROM functionality using the flash memory. The `Fee` is part of the memory services in the microcontroller abstraction layer.

Fibex

Fibex is an XML-based format for the description of complex and message-based communication systems. The Fibex format is especially used to describe FlexRay networks. Besides using the LDF description file format, it is possible to define LIN networks with the Fibex format, too.

filter

Value-based filters can be applied to primitive data element prototypes with data semantics (`isQueued =FALSE`) as part of a data receiver Com specification in a require port to filter the data on receiver side, i.e. the data is not passed to the application if the filter does not match. Filtering is not provided by the `Rte` but is handled by Autosar Com. In order to use the filter mechan-

ism, the appropriate reception filters have to be specified in the Autosar `Com` module for the Com signal(s) that is used to receive the data element prototype. If the communication is intra-ECU, the data element prototype has to be mapped to internal Com signal(s) in order to use the filter mechanism. The Rte generator does not check whether the filter specification in the data receiver Com specification and the filter specification for the corresponding Com signal(s) are consistent.

`FiM` — `Function Inhibition Manager` module: The `Function Inhibition Manager` is responsible for providing a control mechanism for software components and the functionality therein.

`Fls` — `Flash Driver` module: The `Flash Driver` provides services for reading, writing and erasing flash memory and a configuration interface for setting/resetting the write/erase protection if supported by the underlying hardware. The `Flash Driver` is part of the microcontroller abstraction layer. It is the lowest layer of the memory stack.

`Fr` — `FlexRay Driver` module: The `FlexRay Driver` hides the peculiarities of a specific FlexRay controller by providing a device independent (but FlexRay controller-specific) API to the upper layers. The `FlexRay Driver` does not have an own execution context (e.g. `MainFunction`) nor a state machine. The `FlexRay Driver` is part of the communication drivers in the microcontroller abstraction layer.

`FrIf` — `FlexRay Interface` module: The `FlexRay Interface` provides equal mechanisms to access a FlexRay bus channel regardless of its location (microcontroller-internal or -external). It abstracts from the location of CAN controllers (on-chip or on-board), the ECU hardware layout and the number CAN drivers. The `FlexRay Interface` is part of the communication hardware in the Autosar ECU abstraction layer

`FrNm` — `FlexRay Network Management` module: The `FlexRay Network Management` is responsible for the cyclic transmission of network management messages on the FlexRay bus. The `FlexRay Network Management` is part of the communication services inside the Autosar service layer. It is positioned below the generic `Network Management` (`Nm`) and uses the `FlexRay Interface` (`FrIf`) to send and receive network management messages.

`FrTp` — `FlexRay Transport Layer` module: The `FlexRay Transport Layer` segments and reassembles messages that do not fit in one of the assigned Fr N-PDUs. The `FlexRay Transport Layer` is located between the `PDU Router` and the `FlexRay Interface` module in the communication services.

`FrTrcv` — `FlexRay Transceiver Driver` module: The `FlexRay Transceiver` is a hardware device, which mainly transforms the logical 1/0 signals of the

microcontroller ports to the bus-compliant electrical levels, currents and timings. Within an automotive environment there is currently only one single physical layer specification for FlexRay. In addition, the transceivers are often able to detect electrical malfunctions such as wiring issues, ground offsets or collisions. Depending on the interface they flag, the detected error is summarized by a single port pin or very detailed via SPI. The `FlexRay Transceiver Driver` is located in the communication hardware abstraction in the abstraction layer.

GCC — GNU compiler collection; originally stood for GNU C compiler (analog to the Unix command cc for C compiler), but has taken on the meaning compiler collection since GCC is able to compile languages other than just C by now.

`Gpt` — `General Purpose Timer Driver` module: Simple driver for creating one-shot/periodic time intervals using a hardware timer. Generates a clock signal. The `Gpt` is part of the microcontroller abstraction layer.

GNU — supposedly stands for *G*NU is *n*ot *U*nix; the GNU General Public License is a free, copyleft license for software and other kinds of works. Developers who use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

`Icu` — `Input Capture Unit Driver` module: The `Icu` is a module that uses the input capture unit (ICU) for demodulation of a pulse-width modulated (PWM) signal. It is counting pulses, measuring frequency and duty cycle, generating simple interrupts and wake-up interruptions. It provides services for: signal edge notification, controlling wakeup interrupts, periodic signal time measurement, edge time stamping (for the acquisition of non-periodic signals) and edge counting. The `Icu` is part of the microcontroller abstraction layer.

implementation — The implementation that refers to an internal behavior describes the properties of the particular implementation such as code type, resource consumption, etc. During configuration of the `Rte`, an implementation has to be selected for each atomic software component type.

init value — For a data element prototype with data semantics (`isQueued =FALSE`), the presence of an init value ensures that valid data is accessed even if no value has been received. An init value can be specified both on receiver and on sender side using a data receiver Com specification of a data sender Com specification. If no init value is specified for each data element prototype with data semantics, a default init value of 0 is used. If Com signals are used for transmission or reception of a data element prototype, the init value that is specified for the data element prototype and the init value that is specified for the corresponding Com signal(s) have to be consistent. The Rte generator does not check whether they match. Moreover, if several data element prototypes share a single Com signal, they all have to use the same init value.

internal behavior	The internal behavior of an atomic software component type describes the component's functionality and internal structure. Only one internal behavior can be specified for each atomic software component type.
inter runnable variable	Inter runnable variables offer a sender/receiver-like communication mechanism for runnable entities within an atomic software component. The `Rte` ensures data consistency for accesses to the inter runnable variables. Inter runnable variables are specified as parts of a software component's internal behavior. Each inter runnable variable has a type and a communication approach (implicit or explicit). Each runnable entity of the same internal behavior can specify read- or write access to any of the inter runnable variables. If access to an implicit inter runnable variable is specified, the `Rte` provides access to a runnable-specific copy of the Inter runnable variable. If access to an explicit Inter runnable variable is specified, the `Rte` provides access to the shared buffer that represents the inter runnable variable. The access is protected against preemption by other runnable entities that have access to the same inter runnable variable.
`IoHwAb`	`I/O Hardware Abstraction` module.
invalid value	The invalid value is specified as part of a data element prototype's data type and is used for the data element invalidation feature. Only primitive types allow to specify an invalid value (Char type, Integer type, Boolean type, Real type or Opaque type).
I-PDU	Interaction layer protocol data unit. PDU used in the Presentation (interaction) Layer (layer 6) of the ISO/OSI-layer model.
`IpduM`	`I-PDU Multiplexer` module: The `I-PDU Multiplexer` module (IpduM) allows the transmission of several PDUs with the same CAN-ID but with different layout. Therefore one byte of the data segment is used as multiplexer byte. On sender side, the `I-PDU Multiplexer` module is responsible for combining appropriate I-PDUs from `Com` to new, multiplexed I-PDUs and sends them back to the `PDU Router`. On the receiver side, it is responsible for interpreting the content of multiplexed I-PDUs taking into account the value of the selector field.
`isQueued`	The `isQueued` attribute describes the semantics of a data element prototype. If the `isQueued` attribute is set to `FALSE` , the data element prototype has *data semantics*, which implies buffered reception with *last-is-best* semantics and a limitation to non-blocking read accesses. If the `isQueued` attribute is set to `TRUE` , the data element prototype is said to have *event semantics*, which implies queued reception. Blockages of read accesses are possible.
`isService`	The `isService` attribute of a sender/receiver or client/server interface indicates that the interface is defined by an Autosar service and therefore belongs to a standardized Autosar interface.

LDF	LIN description file; the LIN description file LDF describes all signals, data packages, and their time order on a LIN network. If several LIN slaves are connected to a LIN network, all signals and frames are described in the LIN description file. Nodes can identify if a frame is relevant for them from the identifier field of the LIN data frame.
lifecycle header file	A static header file of the `Rte`. It defines the `Rte`'s lifecycle API, i.e. the API functions for initializing and finalizing the `Rte`. It is included by BSW modules which use the `Rte`'s lifecycle API.
`Lin`	`LIN Driver` module: The `LIN Driver` performs hardware access and offers a hardware-independent API to the upper layer. Only the `LIN Interface` has access to the `LIN Driver`. It provides services to control the behavior and state of the UARTs serving as LIN hardware unit. The `LIN Driver` is part of the `Com Driver` in the microcontroller abstraction layer. It is the lowest layer of the LIN bundle.
LIN	Local Interconnect Network: Serial network based on time-scheduling on low-speed networks (up to 19.2 kbit/s).
`LinIf`	`LIN Interface` module: The `Lin Interface` implements the core functionality of a LIN master. This includes the processing of LIN schedule tables and therefore the control over which and when frames are transmitted on the bus. It is part of the communication hardware abstraction in the ECU abstraction layer.
L-PDU	PDU used in the Data Link Layer (layer 2) of the ISO/OSI layer model.
µC	Microcontroller
`make`	A `make` file is a software program intended to automate and optimize the construction of programs or files. `Make` is one of the original Unix tools for software engineering. There are public domain versions (e. g. GNU) and versions for other systems (e. g. Vax/VMS).
Makefile	A Makefile provides instructions for a `make` program (this program is called target), and the rules how to implement these instructions. Every target may depend upon other targets or files.
MCAL	See: microcontroller abstraction layer
`Mcu`	`MCU Driver` module: The `MCU Driver` provides services for basic microcontroller initialization, power down functionality, reset and microcontroller-specific functions required for other MCAL software modules. The `MCU Driver` is part of the microcontroller abstraction layer.
MCU	Microcontroller unit; is a single chip that contains a processor, RAM, ROM, clock and I/O control unit.

`MemIf`	`Memory Abstraction Interface` module: The `Memory Abstraction Interface` abstracts how the memory is stored. This enables the `NVRAM Manager` to manage multiple EEPROM drivers via the `Ea` module, and/or multiple flash drivers via the `Fee` module.
microcontroller abstraction layer	Microcontroller abstraction layer (MCAL); abstracts µC periphery and memory from the upper layer software layers of the Autosar. The MCAL provides functions for configuring the processor periphery, writing and reading of special function registers, i.e. Dio, Pwm. The functions for the modules within the MCAL are typically called *drivers*.
mode declaration	A mode declaration specifies a single mode by assigning a name to the mode.
mode declaration group	A mode declaration group aggregates several mode declarations. A mode declaration group prototype is aggregated by a sender/receiver interface and is typed by a mode declaration group.
mode disabling dependency	The mode disabling dependency is a property of an Rte event that triggers a runnable entity. If the Rte event that triggers a runnable entity has a mode disabling dependency for a certain mode declaration, the runnable entity is not triggered by the Rte event as long as the mode specified by the mode declaration is active.
mode manager	The mode manager for a mode declaration group prototype is a software component with a provide port for the sender/receiver interface that aggregates the mode declaration group prototype. This implies that the mode manager can switch between the modes that belong to the mode declaration group prototype.
mode port	A mode port is a provide or require port that is categorized by a sender/receiver interface that aggregates at least one mode declaration group prototype.
mode switch event	A mode switch event is issued by the `Rte` to notify the receiver in a sender/receiver connection for a mode declaration group prototype that a mode switch has occurred. A mode switch event is only allowed to trigger a runnable entity. It is not allowed to resolve a wait point. Two different kinds of activation for mode switch events are distinguished, depending on the mode switch event's activation attribute. If `activation = exit` is specified, the mode switch event triggers a runnable entity when leaving the old mode. If `activation = entry` is specified, the mode switch event triggers a runnable entity when entering the new mode.
mode switch interface	A sender/receiver interface that aggregates at least one mode declaration group prototype is also called a mode switch interface.
mode user	The mode user for a mode declaration group prototype is a software component with a require port for the sender/receiver interface that aggregates the mode declaration group prototype. This implies that the mode user is notified about

	mode switches by the `Rte`, can use mode disabling dependencies on Rte events and can specify mode switch events that trigger runnable entities.
multiple instantiation	An atomic software component type is said to have multiple instances if more than one instances of the atomic software component type exists within the system. Multiple instances of an atomic software component type can be created by specifying more than one component prototype of the atomic software component type within a composition type or by specifying more than one component prototype of the composition type that contains instance(s) of the atomic software component type. Multiple instantiation of an atomic software component prototype is only possible if its internal behavior supports the `MultipleInstantiation` attribute. If that attribute is set, the different instances are distinguished by the instance handle. The instance handle is then always present as first parameter of all Rte API calls that are generated for this atomic software component type, for all `trace hook` functions for those API calls and for all runnable entity prototypes of the atomic software component type's internal behavior. If the `MultipleInstantiation` attribute is not set, the instance handle is omitted as first parameter of all Rte API calls that are generated for this atomic software component type, for all trace hook functions for those API calls and for all runnable entity prototypes of the atomic software component type's internal behavior.
native NV block	The native NV block is a configurable type of an NV block.
`Nm`	`Generic Network Management Interface` module: The `Generic Network Management Interface` module (`Nm`) implements a wrapper layer above the `CanNm` and the `FrNm`, so that the `ComM` only sees one homogeneous Nm layer. It features the forwarding of calls. The `Generic Network Management Interface` is an adaption layer between the Autosar communication manager and the Autosar bus-specific network management modules (e.g. `CanNm`, `FrNm`).
N-PDU	PDU used in the Network Layer (layer 3) of the ISO/OSI layer model.
NV	non-volatile
`NvM`	`NVRAM Manager` module: The `NVRAM Manager` provides services to ensure the data storage and maintenance of non-volatile data according to their individual requirements in an automotive environment. The `NVRAM Manager` administrates the non-volatile data of an EEPROM and/or flash EEPROM emulation device. It provides the required synchronous or asynchronous services for the management and the maintenance of non-volatile data. It is part of the memory services in the service layer.
NVRAM	Non-volatile random access memory.

NVRAM block	The NVRAM block is the entire structure, which is needed to administrate and to store a block of non-volatile data. The NVRAM block is composed of basic storage objects.
operating mode	The `Rte` specification distinguishes two operating modes: *compatibility mode* and *vendor mode*. Compatibility mode guarantees that software components compiled against a contract phase application header file of one Rte generator operating in compatibility mode are compatible with an `Rte` generated by any other Rte generator operating in compatibility mode. Vendor mode enables vendor-specific optimizations, but the same Rte generator has to be used in the contract- and in the generation phase.
operation invoked event	An operation invoked event is issued by the `Rte` to notify the server in a client/server connection that an operation prototype has been invoked. An operation invoked event is only allowed to trigger a runnable entity. It is not allowed to resolve a wait point. runnable entities that are triggered by operation invoked events are also called server runnable entities. They might have a different signature than runnable entities triggered by other Rte events, depending on the argument prototypes of the operation prototype corresponding to the operation invoked event.
Operation prototype	An operation prototype is aggregated by a client/server interface. It represents the operation prototype that is provided/can be invoked in a port that provides/requires the client/server interface. An operation prototype can use any of the possible application errors that are defined in the client/server Interface. The operation prototype can specify argument prototypes that are passed between client and server when the operation prototype in a require port is invoked.
`Os`	`Operating System` module: Autosar `Os` is an OSEK-compatible operating system with several extensions. It features: schedule tables, applications, memory protection (if supported by the processor) and timing protection It is part of the system services.
OS	Operating system
OSEK communication	Uniform communication environment for automotive control unit application software.
PCI	Protocol Control Information; the part of a PDU that contains control information.
PDU	Protocol data unit. Signals are packed into PDUs for transmission to save bus bandwidth. A PDU consists of PCI (Protocol Control Information) and SDU (Service Data Unit). Depending on the layer the PDU carries a prefix, i.e. *I* for Interaction Layer, *N* for Network Layer, *L* for Data Link Layer.
`PduR`	`PDU Router` module: The `PDU Router` (`PduR`) is responsible for routing protocol data units (see also PDU) from the lower layer modules (`Can`, `Lin`,

Fr) towards the upper layer modules Dcm and Com. It also provides gateway functionality by routing PDUs among the various buses (CAN, FlexRay, LIN etc). The PDU Router is par of the communication services in the service layer.

per instance memory

The per instance memory defines an instance-specific block of memory. All runnable entities of a software component's instance can access the per instance memory, but data consistency is not guaranteed for accesses to the per instance memory. The per instance memory is specified as part of an atomic software component's internal behavior. Per instance memory blocks are typed. The data type is part of the per instance memory specification.

port

port

A software component has ports, through which the software component can interact with other software components. A port always belongs to exactly one software component and represents a point of interaction between a software component and other software components.

Port

Port Driver module: The Port Driver initializes all port pins of the CPU. It allows to configure the pin mode (Dio, Adc, Pwm), the pin direction (in/out), the pin level (high/low) and whether the direction is changeable at runtime or not. To configure a port/pin the user has to know the pin number of the CPU package (for example in 208-pin QFP or BGA). This pin number can be assigned to a user-specific symbolic name, so that this name can be used in the software. The Port Driver is part of the microcontroller abstraction layer.

port-defined argument values

Port-defined argument values provide a mechanism to pass implicit arguments to a server runnable that remain hidden for the client. This mechanism is used extensively in client/server communication between a software component and basic software modules (Autosar Services, IO Hardware Abstraction). In order to use port-defined argument values a port argument list has to be specified as part of the server software component's internal behavior. The port argument list refers to the port that provides an operation prototype and contains an ordered list of primitive values that are included by the Rte between the instance handle (if existent) and the operation-specific parameters (argument prototypes) whenever the runnable entity is invoked. In order to ensure that the port-defined argument values are specific for each client software component that is connected to the server software component, the client/server connections have to be 1:1, i.e. a separate provide port is necessary for each client.

provide port prototype

A provide port prototype is part of a software component type (atomic software component type or composition type) and is categorized by an interface. If the interface is a sender/receiver interface, all data element prototypes that belong

to the interface can be sent through the provide port. If the interface is a client/server interface, all operations prototypes that belong to the interface are available for calls in the provide port. If the interface is a mode switch interface, mode switches for all the modes declared in the mode declaration group prototypes of the interface, can be notified through the provide port.

`Pwm`
: `PWM Driver` module: The `PWM Driver` provides services for the initialization and control of the microcontroller internal pulse width modulation stage. The `Pwm` module generates pulses with variable pulse width. It allows the selection of the duty cycle and the signal period time. The `PWM Driver` is part of the microcontroller abstraction layer.

queue length
: A queue length has to be defined for the queued reception of data element prototypes with `isQueued =TRUE` in the event receiver Com specification and for the queueing of server calls of operation prototypes in the server Com specification.

RAM
: Random access memory

RAM block
: The RAM block is a basic storage object. It represents the part of an NVRAM block which resides in the RAM. The RAM block is a mandatory part of an NVRAM block.

`RamTst`
: `RAM Test` module: The `RAM Test` implements a functional test of microcontroller internal RAM cells. It is part of the memory drivers in the microcontroller abstraction layer.

redundant NV block
: The redundant NV block is a configurable type of a NV block.

require port prototype
: A require port prototype is part of a software component type (atomic software component type or composition type) and is categorized by an interface. If the interface is a sender/receiver interface, all data element prototypes that belong to the interface can be received in the require port. If the interface is a client/server interface, all operation prototypes that belong to the interface can be invoked in the require port. If the interface is a mode switch interface, the software component is notified of mode switches for all the modes declared in the mode declaration group prototypes of the interface in the require port and can react accordingly (through mode switch events or mode disabling dependencies).

ROM
: Read-only memory.

ROM block
: The ROM block is a basic storage object. It represents the part of the NVRAM block which resides in the ROM. The ROM block is an optional part of an NVRAM block.

`Rte`
: `Run-Time Environment`: The `Run-Time Environment` (`Rte`) is at the heart of the Autosar architecture. The Autosar Rte provides a hardware-independent communication infrastructure for application software (Autosar software

components) in an Autosar system on the one hand, and a run-time environment for the software component's runnable entities on the other hand. The communication infrastructure provided by the Rte enables communication between Autosar software components as well as communication between Autosar software components and certain basic software modules (Autosar `Services` and `IoHwAb`). The `Rte` is generated for each ECU to meet the specific needs of the application software on that ECU while keeping code and run-time overhead at a minimum. In order to provide its functionality, the Autosar `Rte` uses services of the `Os` and of `Com`. Furthermore the `Rte` connects the software components with the service modules of the basic software (`NvM`, `WdgM`, `ComM`, `EcuM`, `Det`, `Dem`, `Dcm`, `FiM`).

Rte event

Rte events allow an event-based activation of runnable entities. runnable entities can be started in response to an Rte event or they can be woken up at a wait point (not possible for all Rte events). Rte events are part of a software component's internal behavior. The Autosar meta model distinguishes timing events, data received events, data receive error events, data send completed events, operation invoked events, asynchronous server call return events, and mode switch events.

`Rte configuration` header file

The `Rte configuration` header file contains the configuration of the VFB tracing (globally enabled or disabled) and the individual configuration of all enabled VFB tracing events.

`Rte` header file

A static header file of the `Rte`. It defines static information such as version information, vendor and module identification and error codes.

Rte phase

The second of the two generation phases distinguished by an Rte generator. The whole `Rte` is generated in the second phase, including the application header files and Autosar types header files, the `VFB tracing` header file, the Rte configuration header file and the C-code of the generated Rte itself. Atomic software components that are delivered as source-code components are compiled against the application header files that are generated in the generation phase. If atomic software components that are delivered as object-code components were compiled against application header files generated in compatibility mode, the Rte generator has to use compatibility mode for the generation phase as well. The object code of all software components (object-code and source-code components) is linked to the code of all basic software modules and of the `Rte` to build the ECU software.

runnable entity

A runnable entity is part of a software component's internal behavior and serves to structure the software component's functionality. It can be executed and scheduled independently from other runnable entities of the software component. Each runnable entity should be triggered by at least one Rte event, otherwise it will never be executed by the Rte (an exception are runnable entities

	of complex drivers, services and ECU abstraction which might be triggered by external events, e.g. interrupts). A runnable entity is implemented by a C-function. The runnable entity's entry point (`symbol` attribute) specifies the name of the C-function. The runnable entities of each software component Instance have to be mapped onto Os tasks, except server runnables that can be invoked as a direct function call.
Rx-	Stands for receive, e.g. in Rx-PDU.
`SchM`	`BSW Scheduler` module: `BSW Scheduler` is the central integration entity of the Autosar standard core. Hence, this central module provides means to: embed BSW module implementations into the Autosar `Os` context, trigger main processing functions of the BSW modules and apply data consistency mechanisms for the BSW modules. The `BSW Scheduler` is part of the system services and therefore provides services for the periodic activation of functions and locking services of global data for all modules of all layers. The `BSW Scheduler` is part of the system services in the service layer.
SDU	Service Data Unit; the part of a PDU that contains the data.
sensor-actuator software component	A sensor-actuator software component type is a special kind of atomic software component type that interacts with sensors/actuators on the ECU via the `I/O Hardware Abstraction`.
server Com spec	Server Com specification; a server Com specification specifies communication attributes for an operation prototype that can be invoked in a provide port. For an operation prototype, the attribute `queue length` (mandatory) has to be specified.
sender/receiver Com specification	A sender/receiver Com specification specifies communication attributes for a data element prototype that is received by a component in a require port or sent by a component in a provide port. For a queued sender data element prototype, the attribute `transmission acknowledgement` (optional) can be specified. For a queued receiver data element prototype, the attribute `queue length` (mandatory) must be specified. For an unqueued data element prototype of a require port, the attribute `init value` must be specified.
sender/receiver interface	A sender/receiver interface aggregates all data element prototypes that are sent/received through a provide/require port and all mode declaration group prototypes that can be notified in a mode port (mode switch interface).
server runnable entity	A runnable entity that implements an operation prototype provided in a provide port with a client/server Interface. A server runnable entity is triggered by an operation invoked event. A single server runnable entity can implement an operation prototype that is provided in several provide ports. Under certain circumstances a server runnable entity does not have to be mapped to a task

	but can be invoked as a direct function call, which means that it is executed in the task context of the client.
service module	A service is a basic software module which offers services to other BSW modules or to the application such as the `ECU State Manager`, the `NVRAM Manager` , the `Com Manager`, the `Diagnostic Event Manager`, the `Development Error Tracer`, `Diagnostic Communication Manager`, the `Function Inhibition Manager`, or the `Watchdog Manager`. The interfaces of the service modules to other basic software modules are implemented as standard C API, whereas the communication with the application is performed through standardized Autosar interfaces via the `Rte` (client/server or sender/receiver communication). So from the application's point of view, the service module is a software component. This means a software component description of the service module must be available. The ports of the application software component instances and the service component prototypes must be connected by service connector prototypes.
service component	A service component is a special type of an atomic software component. A service component type specifies the external view of an Autosar service. A service component prototype is the instance of a service component type. In contrast to application component prototypes, service component prototypes are not instantiated within a composition type but within a special ECU software composition.
software component	This term is often used as a synonym for atomic software components, sensor-actuator software components or composite components. A software component encloses software that realizes a particular functionality. Communication with other software components is possible through ports (external view, described by the software component's type). Its functionality is provided by runnable entities (internal view, described by the software component's internal behavior). Autosar allows a hierarchical structure of software components. Atomic software components are the smallest non-dividable software entities. Sensor-actuator software components are special atomic software components that control a sensor or actuator on the ECU.
software component description	SW-CD; one or more XML files which describe the architecture, e.g. port, of the software components. The format is standardized by Autosar.
`Spi`	`SPI Driver` module: The `SPI Driver` provides services for reading from and writing to devices connected via SPI buses. It provides access to SPI communication for several users. It also provides the required mechanism to configure the on-chip SPI peripheral. The `SPI Driver` is part of the communication drivers in the microcontroller abstraction layer.
SW-C	See software component.
SW-CD	See software component description.

synchronous server call point

Property of a runnable entity that indicates that the runnable entity synchronously invokes an operation prototype in the provide port of a client/server interface by using the appropriate Rte API call. Synchronous invocation of an operation prototype implies that the Rte API call blocks until the execution of the server has terminated.

system description

One or more XML files which describe a system. The system description includes

- hardware topology
- communication matrix
- software architecture
- software component to ECU mapping
- data mapping

Typically, the system description contains all software component descriptions of the system. The format is standardized by Autosar.

timing event

A timing event is an Rte event that allows periodic activation of a runnable entity. Its period is specified as an attribute of the timing event. A runnable entity cannot have a wait point for a timing event.

trace event

Element of the VFB tracing mechanism provided by the `Rte`. Trace hook functions are provided for each trace event. The `Rte` distinguishes the following trace events:

- Rte API start (when an Rte API call is made)
- Rte API return (before an Rte API call returns)
- signal transmission (before Com_SendSignal or Com_UpdateShadowBuffer is invoked)
- signal reception (after Com_ReceiveSignal or Com_ReceiveShadowBuffer has returned)
- Com callback (when a Com callback function is invoked)
- task activate (was invoked before a task containing runnable entities is activated)
- task dispatch (invoked when an Rte-generated task begins execution)
- set Os event (invoked before Rte code sets an Os event)
- wait Os event (invoked before Rte code invokes the WaitEvent Os service)
- received Os event (invoked after Rte code had returned from waiting for an Os event)
- runnable entity invocation (invoked before runnable entity starts execution)

- runnable entity termination (invoked after runnable entity has terminated execution).

trace hook function	Function that is automatically invoked by the `Rte` when the corresponding trace event occurs (if trace event enabled). No implementation is generated by the Rte generator for the trace hook functions. The user must provide the implementation.
transmission acknowledgement	Transmission acknowledgement is specified as part of the sender Com specification. If transmission acknowledgement is specified for a data element prototype, the sender of the data element prototype is notified whether the new value has been sent successfully or not.
Tx-	Stands for transmit, e.g. Tx-PDU.
VFB	Virtual functional bus; Autosar concept that describes the the top level communication behavior. The `Rte` implements the VFB.
VFB tracing	VFB tracing is a mechanism provided by the `Rte` that allows to monitor the interaction of software components with the VFB.
VFB tracing header file	The `VFB tracing` header file contains the prototypes of the trace hook functions for the trace events that are configured in the `Rte configuration` header file. the `VFB tracing` header file is included by the generated Rte and must be included in the module(s) that define the configured trace hook functions.
wait point	Property of a runnable entity that indicates that the runnable entity waits for the occurrence of an Rte event or for a timeout to expire. Not all Rte events can resolve a wait point. Wait points are restricted to asynchronous server call returns events, data received events and data send completed events.
`Wdg`	`Watchdog Driver` module: The `Wdg` provides services for initialization, changing the operation mode and triggering the watchdog. The functional requirements and the functional scope are the same for both internal and external watchdog drivers. Hence the API is semantically identical. The `Watchdog Driver` is part of the microcontroller abstraction layer.
`WdgIf`	`Watchdog Interface` module: The `Watchdog Interface` abstracts from the `Watchdog Driver`(s) to make the `WdgM` access to the lower layers hardware (ECU) independent. The `Watchdog Interface` is an on-board device abstraction.
`WdgM`	`Watchdog Manager` module: The task of the `Watchdog Manager` is to monitor the correct operation of system components (called *entities* in the SWS) and to trigger a hardware watchdog on a periodic basis. Different operating modes can be configured: The `Watchdog Manager` can monitor a

	single entity, multiple entities or no entity at all. It is executed periodically. The `Watchdog Manager` is part of the service layer.
XML	Extensible *M*arkup *L*anguage

Index

A

About EB tresos Studio, 48
Autosar format, 38

B

Bundle Registry, 83

C

CAN network, 222
CanNm, 196
CanTp, 194
Com, 188, 191
COM configuration, 235
Com configuration, 239
COM container, 235
Com container, 239
Command line interface, 274
Command shell, 274
Configuration and generation tool
  EB tresos Studio, 29
Configuration data
  Export, 177
  Import, 177
Content type, 37, 177

D

DAD files, 219
DBC files, 219
DBC Importer, 201, 207-208, 214, 218
  Attaching attributes, 201
  Attributes, 201
  Messages, 201
  Multiplexed messages, 201
  NM messages, 201
  Signals, 201
DBC Parser
  grammar, 224

E

EB tresos Studio verifier, 71
ECU Extract
  Importing, 179
EcuC configuration, 237, 241
Editor view, 60-62
Element Description, 64
Element info
  Parameter meta information, 63
Element Outline view, 60
Error Log, 65, 159-160
  Filtering messages, 68
Error Log view, 60
  Log entries, 67
Export
  Autosar, 177

F

Fibex Importer, 242, 247-248, 259, 265
File type, 37
FlexLM Licenses, 92
Floating Network Licenses, 92

G

Global PDU, 237, 241
Graphical user interface, 41
GUI, 41
  Configuring a module, 132
  Editor view
    Choices, 140
    Label, 136
    Optional elements, 144
    References, 139
    Tables, 141
    Textbox, 136
    Type icon, 136
  Element Description
    Parameter, 64
  Element info
    Parameter meta information, 63

H

Help menu, 48

I

Import
  Autosar, 177
importer
  naming rules, 271
Importing system descriptions (Sys-D)
  Adding a System Description Importer, 180

L

LDF files, 227
LDF format, 227
LDF Importer, 227, 232
Licenses, 92
LIN configuration, 227
LIN Driver, 233, 237
Log files, 36

M

Module Registry, 82

N

Network Licenses, 92
Node Outline view, 60
  Editor contents, 61
  Filter displayed items, 61

O

OIL files, 197
OIL format, 197
OIL Importer, 197
Os configurations, 197
Os module, 197

P

Parameter, 64
  (see also Element Description)
Parameter meta information, 63
  (see also Element info)
PDU Router, 187, 191, 212, 217, 255, 262
PDU router, 234, 239
Plugin Registry, 83
Problems view, 60, 71, 159-160
  Filtering messages, 73
Project
  Delete, 109
  Export, 166
  Generate code, 158
  Import, 161
project
  Convert to newer version, 127
Project configuration data
  Exporting, 170
  Importing, 170
Project Explorer, 107-108, 110
  Hide layers, 52
Project Explorer view, 60
Project properties, 112
  Creation of child elements, 116
  Generation path, 114
Project Wizard, 99
Prorietary format, 224

S

Search, 153
Store project as compressed archive, 169
Synchronizing Project Explorer with Editor view, 58
system description
  Importing, 179
System Description Importer, 186

T

TDB Importer, 266
Temporarily disable configurations, 125
Tresos Details dialog, 79

U

Unix line endings, 114
Use of OIL Importer, 200

V

Valid C-Identifiers, 271
Vendor-specific module definition, 295, 299, 301

W

Workspace, 36