

IAR visualSTATE[®]

User Guide

COPYRIGHT NOTICE

© Copyright 2002–2008 IAR Systems AB.

No part of this document may be reproduced without the prior written consent of IAR Systems. The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such a license.

DISCLAIMER

The information in this document is subject to change without notice and does not represent a commitment on any part of IAR Systems. While the information contained herein is assumed to be accurate, IAR Systems assumes no responsibility for any errors or omissions.

In no event shall IAR Systems, its employees, its contractors, or the authors of this document be liable for special, direct, indirect, or consequential damage, losses, costs, charges, claims, demands, claim for lost profits, fees, or expenses of any nature or kind.

TRADEMARKS

IAR Systems, IAR Embedded Workbench, C-SPY, visualSTATE, From Idea To Target, IAR KickStart Kit, IAR PowerPac, IAR YellowSuite, IAR Advanced Development Kit, IAR, and the IAR Systems logotype are trademarks or registered trademarks owned by IAR Systems AB. J-Link is a trademark licensed to IAR Systems AB.

Microsoft and Windows are registered trademarks of Microsoft Corporation.

Unified Modeling Language and UML are registered trademarks or trademarks of the Object Management Group, Inc.

Borland is a registered trademark, and Delphi is a trademark of Borland Software Corporation.

Altia is a registered trademark of Altia, Inc.

All other trademarks and registered trademarks are the property of their respective owners.

EDITION NOTICE

Fourth edition: November 2008.

This document applies to version 6.2 of the IAR visualSTATE software.

Part number: UVS-4.

Contents

Figures	xiii
Tables	xxi
Preface	xxiii
Structure of this guide	xxiii
Assumptions and conventions	xxvi
Part I: Introduction	1
What is visualSTATE?	3
visualSTATE modules	3
visualSTATE Project examples	4
Sample code	5
visualSTATE user documentation	5
Application development with visualSTATE	7
General	7
visualSTATE APIs	8
Code required for a visualSTATE application	9
Getting started	11
How you start visualSTATE	11
Setting up a visualSTATE Project	12
Part 2: Project management	15
Graphical environment	17
General	17
Navigator windows	18
Navigator toolbars	19
Customizing the Navigator	20

Handling visualSTATE Projects, Systems and files	21
The workspace	21
Creating and saving a workspace	22
Opening a workspace	25
Creating a new Project in a workspace	25
Adding an existing Project to a workspace	27
Removing a Project from a workspace	28
Setting a Project or System as active	28
Setting Verificator, Coder and Documenter options	29
Reloading files in the Navigator	32
Digital signature	33
Handling Projects from previous visualSTATE versions	35
Closing the Navigator	35
Source code control	37
Supported visualSTATE file types	37
Using source code control	37
User name for source code control system	39
Custom commands	41
What is a custom command?	41
Creating custom commands	41
Activating custom commands	44
Editing, renaming, and deleting custom commands	44
Renumbering of custom command macros	45
Part 3: Modeling	47
Graphical environment	49
General	49
Designer windows	50
Designer toolbars	53
Getting started	57
Designing statechart diagrams	57

Navigating in statechart diagrams	60
Resizing and positioning objects in statechart diagrams	63
Printing statechart diagrams	63
Safe mode	64
Customizing the Designer	64
States	67
Composing states	67
Composite states	72
Regions	75
Connector states	77
Pseudostates	77
Excluding states and regions	80
Transitions	83
Composing transitions	83
Completion transitions	86
Elements	89
Creating and editing elements	89
Searching for an element	95
Handling Projects, Systems, and files for modeling	97
Creating and saving Projects, Systems, and files in the Designer	97
Opening a Project in the Designer	101
Importing files into the Designer	101
Specifying number of System instances	101
Using Designer backup files	102
Using function declarations and constants in existing files	104
Closing the Designer	106

Part 4: Formal testing	107
Introduction	109
Conventions used in this part	109
Verification with visualSTATE Verificator	110
Overview	110
Approach	112
Aspects of formal verification	113
Checks performed by visualSTATE Verificator	123
Check for unused elements	123
Check for activation of elements	125
Check for conflicting transitions	128
Check for state dead ends	129
Check for local dead ends	130
Check for System dead ends	131
Check for dynamic ambiguous assignments	131
Check for static ambiguous assignments	133
Check for signal queue size	133
Overview of checks, modes, and errors	135
Verifying your visualSTATE Project	137
Starting verification	137
Tracing your visualSTATE Project	141
Performing a trace	141
Designing for verification	143
Using time/memory options to help verification	143
Keeping down the complexity of verifying Systems	144
Verification and visualSTATE generated code	146

Part 5: Functional testing	147
Introduction	149
Simulation with visualSTATE Validator	149
Graphical environment	150
Simulation	161
Starting simulation	161
Sending events	162
Viewing elements during simulation	164
Specifying event parameters	167
Signal queue handling	167
Breakpoints	169
Changing variable values	176
Setting action function return values	177
Forcing states	177
System setup	178
Graphical animation	179
Toggling between Validator mode and target mode	180
Tracing visualSTATE models	183
Tracing	183
Recording and playing test sequences	187
Recording a test sequence	187
Playing recorded test sequences	191
Analyzing visualSTATE models	195
Static analysis	195
Dynamic analysis	197
Part 6: Testing in target applications	201
Introduction	203
What is RealLink?	204

RealLink connection to target	204
visualSTATE elements supported by RealLink	205
Target requirements	206
Testing visualSTATE models using RealLink	207
Setting up RealLink	207
Monitoring your target application	220
Controlling your application in target	224
Recording and playing sequences of target tests	227
Troubleshooting	228
 Part 7: Code generation	 231
Introduction	233
Code generation and visualSTATE APIs	233
Description of generated code	234
Real-time operating system (RTOS)	235
Generating code	237
Starting code generation	237
Generating C++ code	237
Basic API code generation	239
Description of generated code	239
Default table-based code configuration	243
Expert API code generation	245
Description of generated code	245
Default configuration	247
Size of generated code	249
Data width	249
Rule data formats	250
Coder options	251
Code size using visualSTATE	251
The size of human-readable code	253

Part 8: Documenting visualSTATE Projects	255
Introduction	257
Project report	257
Creating a Project report	258
Viewing the Project report	259
Setting up a visualSTATE Project report	261
General	261
Specifying report contents	262
Specifying report output format	266
Setting up standard report layout	268
Customizing report layout	271
Part 9: Prototyping	275
Introduction	277
Prototyping with Altia	279
Basic concepts	279
Interfacing a visualSTATE model to an Altia design	281
Simulation with Altia	285
Closing the Altia connection	286
Using parameters	286
Configuring the Altia connection	288
Prototype based on visualSTATE generated code	291
General	291
Example: Implementing visualSTATE code in C++ code	292
Prototyping with the visualSTATE Expert DLL	299
What is visualSTATE Expert DLL?	299
Interaction	300
Generating code for the visualSTATE Expert DLL	301
Interfacing to the Expert DLL using Visual Basic	302

Part 10: Working in an OSEK environment	311
Using the visualSTATE OSEK Kit	313
Generating visualSTATE files for use in an OSEK environment	313
Enabling OSEK support	313
Assigning visualSTATE Systems to OSEK tasks	315
Building a runtime application	321
Requirements for building a runtime application	321
Exported visualSTATE OSEK API functions	323
Supplying events	323
API examples	324
Runtime considerations	329
Stack usage	329
RAM/ROM usage	333
Part 11: General reference	335
Navigator menu commands	337
File menu	337
Edit menu	339
View menu	339
Project menu	339
Tools menu	340
Window menu	340
Help menu	340
Designer shortcuts	341
General	341
Diagram tools	341
Project, System and statechart diagram views	342
Element browser	344

Designer menu commands	345
File menu	345
Edit menu	346
View menu	347
Insert menu	349
Format menu	350
Tools menu	352
Window menu	353
Help menu	353
Validator shortcut keys	355
General	355
Windows	355
Editing	355
Debugging	356
Navigation in test sequence files	356
Validator menu commands	357
File menu	358
Edit menu	360
View menu	362
Debug menu	363
RealLink menu	366
Altia menu	367
Window menu	368
Help menu	369
Verificator command line options	371
General	371
Command line syntax	371
List of Verificator command line options	372
Coder options	375
Command line syntax	375
Lists of Coder options	376

Documenter options	393
Command line syntax	393
Lists of Documenter options	393
Appendix A: visualSTATE file name extensions	407
Appendix B: RealLink memory consumption	409
visualSTATE model dependent memory usage	409
RealLink API dependent memory usage	410
Appendix C: Source code example	411
Mobile phone.frm	411
Main.bas	423
Utility.bas	429
Appendix D: Handling visualSTATE files from previous versions	435
Manual conversion from format 1 to 6 format	435
Index	437

Figures

1: Example of a Navigator workspace with a visualSTATE Project	7
2: Use of visualSTATE API in a visualSTATE embedded application	8
3: Navigator Project menu	11
4: Navigator opening dialog box	12
5: Navigator New dialog box	12
6: Designer application with newly created Project	13
7: Navigator reload message	14
8: Workspace created in the Navigator	14
9: Navigator, with workspace loaded	17
10: Navigator Properties window	19
11: Navigator Standard toolbar	19
12: Navigator Internet browser toolbar	20
13: Navigator Settings dialog box	20
14: System(s) dialog box, Navigator	23
15: Topstate(s) dialog box, Navigator	23
16: Topstate Region(s) dialog box, Navigator	24
17: New dialog box, Project tab (Navigator)	26
18: Workspace browser with Project	27
19: Insert visualSTATE Project dialog box	28
20: Setting a visualSTATE Project as active	29
21: Coder Project Options dialog box, Configuration tab	30
22: Display of online help	31
23: Navigator reload message	32
24: Navigator Settings dialog box	32
25: Verificator notification	34
26: Conversion of Project from previous visualSTATE version	35
27: Navigator Settings dialog box	39
28: Custom commands dialog box (Navigator)	42
29: Custom commands, arguments pop-up menu (Navigator)	43
30: Custom commands, Select Project dialog box (Navigator)	43
31: Navigator workspace with custom command	44

32: Designer environment with visualSTATE Project	49
33: State pop-up menu	50
34: Designer diagram window	51
35: Designer element browser window	52
36: Designer property window	52
37: Designer output window	53
38: Designer Standard toolbar	53
39: Designer Diagram toolbar	54
40: Designer Size toolbar	54
41: Designer Source Control toolbar	54
42: Designer Zoom toolbar	55
43: Designer with Project loaded	57
44: Newly drawn states	58
45: Examples of transitions	59
46: Designer zoom view, focus on upper left part of statechart diagram	61
47: Designer zoom view, focus on lower right part of statechart diagram	62
48: Objects selected	62
49: Designer Page Setup dialog box	64
50: Designer Customize dialog box, transition category selected	65
51: Compose State dialog box	68
52: Compose State dialog box, Event1 added	69
53: List of elements (Designer)	70
54: New Event dialog box (Designer)	71
55: Compose State dialog box, event created and used as trigger	72
56: Composite state with one region	73
57: Composite state with two concurrent regions	73
58: Selection of states to be moved (Designer)	74
59: Composite state consisting of mutually exclusive substates	74
60: Example of state with one region	75
61: Off-page state region	76
62: System view pop-up menu (Designer)	76
63: Example of a pair of connector states	77
64: Connector state pop-up menu	77
65: Example of a state with an initial state	78

66: Example of fork and join states	79
67: StatePopup	80
68: StateExclusion	80
69: Compose Transition dialog box	84
70: New Event dialog box (Designer)	85
71: Compose Transition dialog box, event created and used as trigger	86
72: Completion transition selected	87
73: Designer element browser, with event created (local element)	90
74: Defining action function	91
75: Compose Transition dialog box, action function	92
76: Define Action Function Parameters dialog box	92
77: External C file specified for action function	93
78: Compose Transition dialog box, guard expression value	94
79: New dialog box	98
80: Designer with blank Project	99
81: Diagram window, with empty statechart diagram	100
82: Compose System dialog box	102
83: Settings dialog box, file backup options (Designer)	103
84: Import Elements dialog box (Designer)	105
85: visualSTATE System consisting of two state machines, R0 and R1	111
86: Example of a System with a large state space.	113
87: Model, interface, and environment.	114
88: Full verification mode, assumptions.	115
89: Guard verification mode, arbitrary values of variables between microsteps	116
90: Guard verification mode, fixed values of variables	116
91: Guard verification mode, assumptions.	117
92: Basic verification mode, assumptions.	118
93: System with ambiguous behavior because of assignments.	119
94: System with ambiguous behavior because of assignments.	120
95: Three Systems of which a and b have ambiguous behavior because of assignments.	121
96: Systems with conflicting transitions.	122
97: System with unused elements	124
98: System with never activated elements	127

99: System with conflicting transitions	128
100: System containing a state dead end.	129
101: System containing a local dead end.	130
102: System containing a System dead end.	131
103: System containing dynamic ambiguous assignments.	132
104: System with two transitions having ambiguous assignments.	133
105: System for which the size of the signal queue must be at least one.	134
106: System which cannot be fully verified.	135
107: Verificator Options dialog box, General tab	137
108: Verificator dialog box	138
109: Verificator notification	139
110: Verification progress window, Navigator	139
111: Verificator Results, Ready to Find Trace	141
112: Specifying trace output file name	142
113: System with deep state space.	145
114: System with shallow state space.	145
115: Validator environment with workspace loaded	150
116: Validator workspace, customized window setup	151
117: New Validator workspace dialog box	152
118: System window (Validator), with pop-up menu	153
119: Event window (Validator), with pop-up menu	153
120: Action window (Validator)	154
121: Variable window (Validator), with pop-up menu	155
122: Guard Expression window (Validator)	155
123: Signal Queue window (Validator), with pop-up menu	156
124: Field Chooser window for Variable window (Validator)	156
125: Validator output window	157
126: Validator Watch window with elements added	157
127: Validator Timers window, with pop-up menu	158
128: Validator Breakpoints window	158
129: Validator Standard toolbar	159
130: Validator Debug toolbar	159
131: Validator RealLink toolbar	159
132: Validator Analysis toolbar	159

133: Initialize Systems dialog box (Validator)	162
134: Validator environment with workspace loaded	163
135: Guard Expression window (Validator)	166
136: Set Event Parameter Value dialog box (Validator)	167
137: Breakpoints Setup dialog box, General tab (Validator)	170
138: Breakpoints Setup dialog box, Events / Signals tab (Validator)	171
139: Breakpoints Setup dialog box, Variables tab (Validator)	172
140: Breakpoints Setup dialog box, Current States tab (Validator)	173
141: Breakpoints Setup dialog box, Action Functions tab (Validator)	174
142: Breakpoint Reached dialog box, Pre-deduct (Validator)	175
143: Breakpoint Reached dialog box, Post-deduct (Validator)	175
144: Variable window (Validator), with pop-up menu	176
145: System window (Validator), with pop-up menu	177
146: System Setup window (Validator)	178
147: Example of graphical animation	179
148: Target command in Validator window	181
149: Trace Setup, Trace To options	183
150: Trace Setup, Trace To Setup	184
151: Trace Point Setup	185
152: Validator Test Sequence File window	187
153: Validator Test Sequence File window, output of selected command	189
154: Pop-up menu of Validator Test Sequence File window	190
155: Test Sequence File dialog box (Validator)	190
156: Log Mismatch Detected dialog box (Validator)	193
157: Validator Analysis toolbar, static analysis	195
158: Validator Static Analysis window, selection of elements to analyze	196
159: Static analysis results (Validator)	197
160: Validator Analysis toolbar (dynamic analysis)	198
161: Validator Dynamic Analysis window, with pop-up menu	199
162: Example of visualSTATE RealLink setup	204
163: RealLink connection between the Validator and target	205
164: Navigator, Coder Options dialog box, Configuration tab	208
165: Navigator, Coder Options dialog box, RealLink tab	209
166: RealLink Properties dialog box	216

167: RS232 Setup dialog box	217
168: TCP/IP Communication Setup dialog box	218
169: Connecting to RealLink	219
170: Validator output window	220
171: Validator Event window in target mode	221
172: Validator Watch window containing visualSTATE elements	222
173: Editing a variable in the Watch window	223
174: Microstep and macrostep in visualSTATE	225
175: Validator RealLink menu commands	226
176: RealLink communication error message	228
177: visualSTATE layers	234
178: Navigator, Coder Options dialog box, Configuration tab	238
179: Enabling human-readable code generation	241
180: Basic API, default configuration	243
181: Expert API, default configuration	247
182: Files that can be included in a visualSTATE Project report	258
183: Documenter Options dialog box, Configuration tab	262
184: Documenter Options dialog box, File Input tab	263
185: Documenter Options dialog box, file inclusion criteria	264
186: Selecting visualSTATE generated files	265
187: Select Files dialog box	265
188: Documenter Options dialog box, File Output tab	266
189: Documenter Options dialog box, Front Page tab	268
190: Documenter Options dialog box, Page Layout tab	269
191: Documenter Options dialog box, Header/Footer tab	270
192: Documenter Options dialog box, Fonts tab	270
193: Documenter Options dialog box, RTF Styles tab	271
194: Documenter Options dialog box, HTML Styles tab	272
195: Altia application loaded with the AVSystem design	280
196: Validator Altia Connect commands	282
197: Open Altia Design dialog box (Validator)	282
198: Validator output window, Altia tab	283
199: Binding Altia objects to visualSTATE elements	284
200: Define Altia Parameters dialog box, Event tab (Validator)	288

201: Define Altia Properties dialog box (Validator)	289
202: Prototype implementation	292
203: visualSTATE statechart	293
204: Visual C++ dialog box	293
205: Prototype implementation, visualSTATE Expert DLL	299
206: Main flow of information, Expert DLL	301
207: Coder Project Options dialog box, Configuration tab	301
208: Mobile phone example	302
209: Navigator Settings dialog box, OSEK page	314
210: OSEK support enabled	314
211: OSEK wizard, first page	315
212: OSEK wizard, Select Systems	316
213: visualSTATE System assigned to an OSEK task	317
214: OSEK wizard, Select runtime options	318
215: OSEK wizard, Summary	319
216: Components required for a runtime application	322
217: Designer Edit menu	346
218: Designer View menu	348
219: Designer Insert menu	350
220: Designer Format menu	350
221: Alignment menu commands, Designer Format menu	350
222: Size menu commands, Designer Format menu	351
223: Space menu commands, Designer Format menu	351
224: Designer Tools menu	352
225: Safe Mode menu commands, Designer Tools menu	352
226: Designer Window menu	353
227: Designer Help menu	353
228: Validator File menu	358
229: Validator Edit menu	360
230: Validator View menu	362
231: Validator Debug menu	364
232: Validator RealLink menu	366
233: Validator Altia menu	367
234: Validator Window menu	368

235: Validator Help menu 369

Tables

1: Typographical conventions used in this guide	xxvi
2: Short forms used in this guide	xxvii
3: Conventions used for constructs	109
4: Verificator checks, modes and errors	136
5: Commands that can be recorded to a Validator test sequence file	188
6: Coder-generated SEM type definitions	249
7: Rule data formats	250
8: Project report sections	262
9: Exported OSEK API functions	323
10: Stack usage by Basic API	329
11: Stack usage by Expert API	329
12: Type sizes determined by runtime application size	330
13: Type sizes determined by compiler, linker and target hardware	330
14: Assumptions for stack size calculation	331
15: Typical stack sizes, Basic API	331
16: Typical stack sizes, Expert API	332
17: Navigator File menu commands	337
18: Navigator View menu commands	339
19: Navigator Project menu commands	339
20: Navigator Tools menu commands	340
21: Designer File menu commands	345
22: Designer Edit menu commands	347
23: Designer View menu commands	348
24: Designer Format menu commands	351
25: Designer Tools menu commands	352
26: Validator File menu commands	358
27: Validator Edit menu commands	361
28: Validator View menu commands	362
29: Validator Debug menu commands	364
30: Validator RealLink menu commands	366
31: Validator Altia menu commands	367

32: Validator Window menu commands	368
33: Verificator command line options	372
34: Configuration project options	377
35: File output project options	378
36: Code project options	379
37: Style project options	382
38: Extended keyword project options	382
39: RealLink project options	383
40: C-SPYLink project options	384
41: API functions project options	384
42: Basic system options	386
43: File output system options	386
44: Code system options	387
45: Readable code system options	389
46: Style system options	389
47: Extended keywords system options	389
48: Names system options	390
49: API functions system options	392
50: RealLink memory consumption, IAR SH7740 32-bit compiler	410

Preface

Welcome to the visualSTATE User Guide.

This guide describes how to use the visualSTATE software for developing and testing embedded applications based on statechart diagrams.

For installation information, see *IAR visualSTATE Installation Guide*.

Structure of this guide

This guide consists of the following parts:

Part 1: Introduction

- *What is visualSTATE?*, page 3, gives a general description of the visualSTATE software and its modules. The chapter also lists the visualSTATE user documentation.
- *Application development with visualSTATE*, page 7, describes the steps involved in a typical visualSTATE development project. It also describes how to use the visualSTATE APIs, and the code required for a visualSTATE application.
- *Getting started*, page 11, describes how to set up a visualSTATE Project and start designing statechart diagrams.

Part 2: Project management

- *Graphical environment*, page 17, describes the graphical environment of the visualSTATE Navigator, which you use for handling visualSTATE files.
- *Handling visualSTATE Projects, Systems and files*, page 21, describes how to handle visualSTATE Projects and files in the visualSTATE Navigator workspace.
- *Source code control*, page 37, describes how to use source code control for your visualSTATE files.
- *Custom commands*, page 41, describes how to set up user-specified commands in a Navigator workspace.

Part 3: Modeling

- *Graphical environment*, page 49, describes the graphical environment of the visualSTATE Designer.
- *Getting started*, page 57, describes how to get started designing statechart diagrams in visualSTATE Designer.
- The chapters *States*, page 67, and *Transitions*, page 83, describe how to create and edit states and transitions in the visualSTATE Designer.

- *Elements*, page 89, describes how to define, create, rename, and delete elements for state reactions and transitions in the Designer.
- *Handling Projects, Systems, and files for modeling*, page 97, describes how create and save new visualSTATE Projects in the Designer, import visualSTATE Systems to a Project, and use Designer backup files.

Part 4: Formal testing

- *Introduction*, page 109, explains what is understood by verification in visualSTATE, and why you are recommended to use it in your development process. It describes the most important concepts related to formal verification, and gives examples of the checks that can be performed by the visualSTATE Verifier.
- *Checks performed by visualSTATE Verifier*, page 123, gives a detailed description of the Verifier checks.
- *Verifying your visualSTATE Project*, page 137, describes how to start verification.
- *Designing for verification*, page 143, gives guidelines on how to design visualSTATE Systems that are to be verified.

Part 5: Functional testing

- *Introduction*, page 149, gives an introduction to simulation with visualSTATE Validator.
- The chapters *Simulation*, page 161, *Recording and playing test sequences*, page 187, and *Analyzing visualSTATE models*, page 195, describe how to use the simulation, debug, and analysis tools of the visualSTATE Validator.

Part 6: Testing in target applications

- *Introduction*, page 203, gives an introduction to visualSTATE Reallink. The chapter describes the Reallink connection to target, visualSTATE elements supported by Reallink, and target requirements
- *Testing visualSTATE models using Reallink*, page 207 describes how to use the Validator Reallink for monitoring and controlling the runtime behavior of a visualSTATE model in a target application.

Part 7: Code generation

- *Introduction*, page 233, gives an introduction to code generation with visualSTATE.
- *Generating code*, page 237, describes how to automatically generate code for visualSTATE models created in the visualSTATE Designer.
- *Basic API code generation*, page 239, describes code generation with the visualSTATE Basic API.
- *Expert API code generation*, page 245, describes code generation with the visualSTATE Expert API.

- *Size of generated code*, page 249, describes how data width and rule data formats influence the size of the visualSTATE generated code.

Part 8: Documenting visualSTATE Projects

- *Introduction*, page 257, describes how to automatically create a visualSTATE Project report.
- *Setting up a visualSTATE Project report*, page 261, describes how to customize a visualSTATE Project report.

Part 9: Prototyping

- *Introduction*, page 277, gives an introduction to prototyping with visualSTATE.
- *Prototyping with Altia*, page 279, describes how you can use the visualSTATE Validator and Altia FacePlace for simulating visualSTATE models.
- *Prototype based on visualSTATE generated code*, page 291, describes how to implement a prototype based on visualSTATE generated code.
- *Prototyping with the visualSTATE Expert DLL*, page 299, describes how to create a visualSTATE prototype using the visualSTATE Expert DLL with Microsoft Visual Basic or C++.

Part 10: Working in an OSEK environment

- *Using the visualSTATE OSEK Kit*, page 313, describes how to enable OSEK support in visualSTATE, and assign visualSTATE Systems to OSEK tasks.
- *Building a runtime application*, page 321 describes how to build a runtime application with ANSI C files generated with the visualSTATE OSEK Kit.
- *Runtime considerations*, page 329 contains information about stack usage and RAM/ROM usage when the OSEK API is used together with the visualSTATE standard APIs.

Part 11: General reference

This part contains an overview of menu commands and shortcut keys in the visualSTATE Navigator, Designer, and Validator. The part also lists options and command line syntax for the visualSTATE Coder, Verificator, and Documenter.

Appendix A: visualSTATE file name extensions

Lists the visualSTATE file name extensions.

Appendix B: RealLink memory consumption

Describes how to how to calculate the additional memory consumption when visualSTATE generated code is used with RealLink.

Appendix C: Source code example

Contains a source code example in Visual Basic.

Appendix D: Handling visualSTATE files from previous versions

Describes how to convert visualSTATE models in visualSTATE version 5 that were created with visualSTATE version 4.x.

Assumptions and conventions

ASSUMPTIONS

This guide assumes that you are familiar with

- The use of Windows-based applications
- Basic principles of state/event modeling
- Programming in C.

Part 10: Working in an OSEK environment assumes that you are familiar with the OSEK standard.

CONVENTIONS

This guide uses the following typographical conventions:

Style	Used for
<i>Italic</i>	Used for emphasis of particular words.
Bold	Refers to window buttons, for example OK .
Xxx>Yyy	Refers to menu commands, for example File>Save As.
CAPITALS	Refers to keys, for example ENTER.
Courier	Used for examples.
TIP	Used for highlighting, for example shortcuts.
Note:	Used for drawing attention to special issues.
<Xxx>.<ext>	This syntax is used for referring to files generated by visualSTATE Coder.

Table 1: Typographical conventions used in this guide

SHORT FORMS

In this guide the following short forms apply:

Short form	Refers to
Navigator	visualSTATE Navigator
Designer	visualSTATE Designer
Verificator	visualSTATE Verificator
Validator	visualSTATE Validator
Coder	visualSTATE Coder
Documenter	visualSTATE Documenter
OSEK Kit	visualSTATE OSEK Kit
Basic API	visualSTATE Basic API
Expert API	visualSTATE Expert API
Expert DLL	visualSTATE Expert DLL
VS Project	visualSTATE Project
VS System	visualSTATE System
Project	visualSTATE Project
System	visualSTATE System

Table 2: Short forms used in this guide

Part I: Introduction

This part of the visualSTATE[®] User Guide includes the following chapters:

- What is visualSTATE?
- Application development with visualSTATE
- Getting started.





What is visualSTATE?

visualSTATE is a Windows-based software package of integrated tools for developing, testing, and implementing embedded applications based on statechart diagrams. It includes a graphical design environment, test tools, a code generator, and a documentation facility.

visualSTATE has been developed in accordance with the Unified Modeling Language notation (UML).

This chapter describes:

- The visualSTATE modules
- visualSTATE Project examples, and how to access them
- Sample code
- visualSTATE user documentation.

visualSTATE modules

The visualSTATE software comprises the following fully integrated modules that allow you to develop and test real-time applications based on statechart diagrams:

- Navigator
- Designer
- Verificator
- Validator, with RealLink
- Coder
- Documenter.

NAVIGATOR

visualSTATE Navigator is a graphics-based project management tool for the overall handling of visualSTATE Projects, from model design over test and simulation to code generation and documentation of visualSTATE Projects. With the Navigator you access and activate the other modules of the visualSTATE software, and set options for the Verificator, Coder and Documenter. For a description of how to use the Navigator, see *Part 2: Project management*, page 15.

DESIGNER

visualSTATE Designer is a graphics-based application for designing statechart diagrams using the UML notation. For a detailed description of how to use the Designer, see *Part 3: Modeling*, page 47.

VERIFICATOR

visualSTATE Verificator is a powerful test tool for dynamic formal verification of models created with the Designer. For a detailed description of verification and how to use the Verificator, see *Part 4: Formal testing*, page 107.

VALIDATOR

visualSTATE Validator is a graphics-based application for simulating, analyzing, and debugging models created with the Designer. With the Validator you can test the functionality of your design. For a detailed description of how to use the Validator, see *Part 5: Functional testing*, page 147.

REALLINK

With the Validator RealLink facility, you can test your visualSTATE model in a target application. See *Part 6: Testing in target applications*, page 201.

CODER

The Coder can automatically generate code on the basis of models created with the Designer. The automatically generated code must be combined with a visualSTATE application programming interface (API) and manually written code (see *visualSTATE APIs*, page 8). For a detailed description of how to use the Coder, see *Part 7: Code generation*, page 231.

DOCUMENTER

With the Documenter you can create an up-to-date documentation report on your visualSTATE Project, including design, tests, and code generation. For a detailed description of how to use the Documenter, see *Part 8: Documenting visualSTATE Projects*, page 255.

visualSTATE Project examples

The visualSTATE software package includes examples of application designs created with visualSTATE. The examples can be used to help fuel your own design as well as provide a reference for design techniques.

The examples can be opened via the Windows Start menu, or the Navigator File menu.

Sample code

The visualSTATE software package includes sample code that you can use as a source of reference in your development projects. The sample code files can be opened via the Examples directory of the visualSTATE software.

visualSTATE user documentation

Installation information is found in *IAR visualSTATE Installation Guide* and visualSTATE installation notes.

You can read more about the visualSTATE software and how to use it in the following user documentation:

- *visualSTATE Quick Start Tutorial* describes how you get started using the visualSTATE software.
- *visualSTATE Concept Guide* describes the basic principles and ideas of the visualSTATE software, and gives a general introduction to the visualSTATE approach and the concept of state machines.
- *IAR visualSTATE Reference Guide* describes the constructs, elements and principles of state machine modeling that are available in visualSTATE. For example it explains constructs such as *states*, *transitions*, *state reactions*, etc.
- *visualSTATE API Guide* describes the visualSTATE APIs and how to use them.

Online versions of the user documentation are included in the visualSTATE software package as PDF files which can be accessed from the visualSTATE Navigator, or the Windows Start menu.

Note: To be able to view the PDF files, you must have Adobe Acrobat Reader installed.

Online help

The visualSTATE Navigator, Designer, and Validator applications offer online help. You activate the online help via the Help menu, or by pressing the F1 key.

To display online help for options in the Navigator settings dialog box and Navigator Project options dialog box, select the option and right-click, or press SHIFT+F1. See example in *Figure 22*, page 31.

The IAR Systems website

If you want to know more about visualSTATE, visit www.iar.com where you will find technical support information, product news, application notes, etc.

Application development with visualSTATE

This chapter describes the steps involved in a typical visualSTATE development project. The chapter also describes how to use the visualSTATE APIs, and the code required for a visualSTATE application.

General

You start a visualSTATE development project by launching the Navigator. In the Navigator you set up your visualSTATE Project in a workspace, including options for verification, code generation, and documentation. See *Part 2: Project management*, page 15.

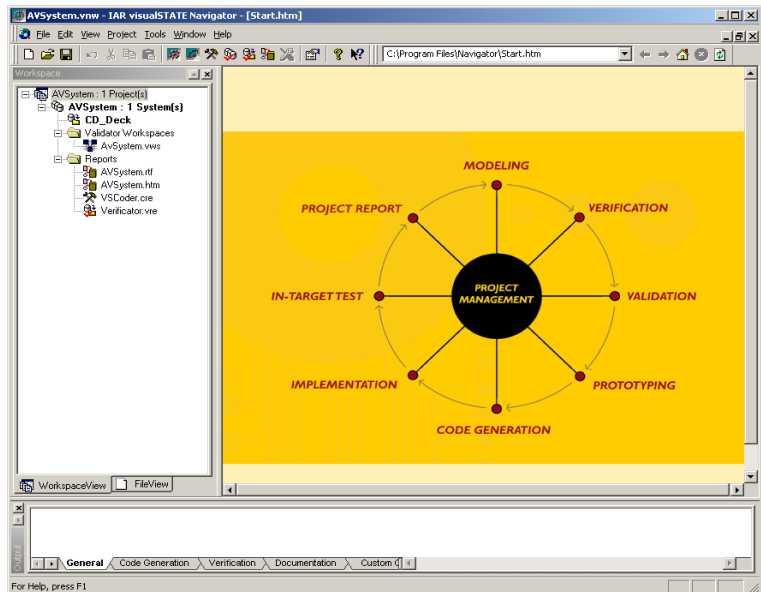


Figure 1: Example of a Navigator workspace with a visualSTATE Project

When you have created the overall structure of your visualSTATE Project, you can start designing visualSTATE models of state machines which is done in the Designer. *Part 3: Modeling*, page 47.

When you have designed your visualSTATE model, you can start testing it. For verification of your visualSTATE model, you use the visualSTATE Verifier. See *Part 4: Formal testing*, page 107.

For interactively simulating, analyzing and debugging the model, you use the visualSTATE Validator. See *Part 5: Functional testing*, page 147.

It is also possible to monitor and control the runtime behavior of visualSTATE models in a target application by means of the Validator RealLink facility. See *Part 6: Testing in target applications*, page 201.

When you have tested your model and corrected it as necessary in the Designer, you can automatically generate the code for it. In target, the code will behave exactly as the model you designed. See *Part 7: Code generation*, page 231.

For documentation of your visualSTATE Project, you can create a documentation report with visualSTATE Documenter. See *Part 8: Documenting visualSTATE Projects*, page 255.

visualSTATE APIs

A visualSTATE API (application programming interface) is a set of files supplied with the visualSTATE software. The visualSTATE API files provide an interface between the visualSTATE Coder-generated code and the user-written code. User-written code is code written by the application developer for communication with the runtime environment.

The use of the visualSTATE API is illustrated in *Figure 2*, page 8.

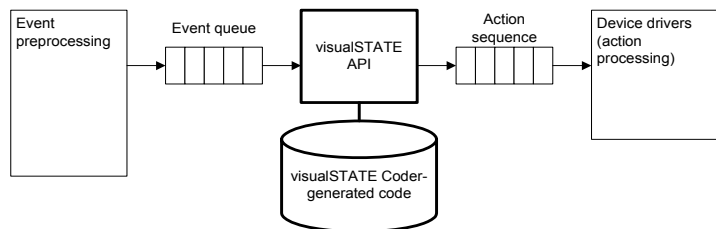


Figure 2: Use of visualSTATE API in a visualSTATE embedded application

Note: In visualState version 5.3 and later, there are two distinct APIs. Figure 2, *Use of visualSTATE API in a visualSTATE embedded application*, describes the Expert API. The Basic API is fully generated by the Coder.

For a detailed description of the visualSTATE standard APIs and how to use them, see *visualSTATE API Guide*.

Code required for a visualSTATE application

In a visualSTATE embedded application, the following categories of code are required:

- visualSTATE Coder-generated code
- visualSTATE API
- User-written code: Manually written code for *event preprocessing*, *event queues*, *device drivers*, *action functions*, and code for calling the functions in the visualSTATE API.

visualSTATE Coder-generated code is code that is generated automatically by the visualSTATE Coder on the basis of statechart designs created in visualSTATE Designer. Before the Coder-generated code is used in target, it must be integrated with the user-written code by means of the visualSTATE API.

Action sequences are handled entirely by visualSTATE. However, the user must write the code for each of the action functions.

This means that the application developer must do the following in order to create a final embedded application using visualSTATE generated code:

- Manually write code for event preprocessing, event queues (if needed), action functions, and device drivers.
- Integrate the user-written code with the Coder-generated code by means of the visualSTATE API.

See *Figure 2*, page 8. See also the sample code included with the visualSTATE software.

For a detailed description of the visualSTATE APIs, refer to *visualSTATE API Guide*.

Getting started

This chapter describes

- How to start visualSTATE and activate the individual visualSTATE programs.
- How you set up a visualSTATE Project in the Navigator.

Installation of visualSTATE is described in *IAR visualSTATE Installation Guide*, and visualSTATE installation notes.

How you start visualSTATE

You start visualSTATE by launching the Navigator via the Windows Start menu (choose Start menu>Programs>IAR Systems>visualSTATE). From the visualSTATE Navigator you can activate all the other visualSTATE programs.

When you have created a workspace in the Navigator (see *Setting up a visualSTATE Project*, page 12), you can launch the other visualSTATE programs and IAR Embedded Workbench® by using the buttons on the Navigator Standard toolbar (see *Figure 11*, page 19), the Navigator Project menu (see *Figure 3*, page 11), or pop-up menu commands.

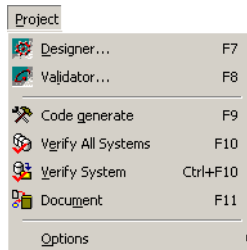


Figure 3: Navigator Project menu

Setting up a visualSTATE Project

You set up your visualSTATE Project in a Navigator workspace (see *The workspace*, page 21), as follows:

- 1 Launch the Navigator via the Windows Start menu. A Navigator opening dialog box is displayed. See *Figure 4*, page 12.

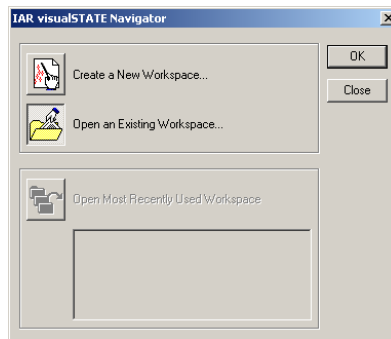


Figure 4: Navigator opening dialog box

- 2 Click **Create a New Workspace** and click **OK**. A dialog box is displayed. See *Figure 5*, page 12.

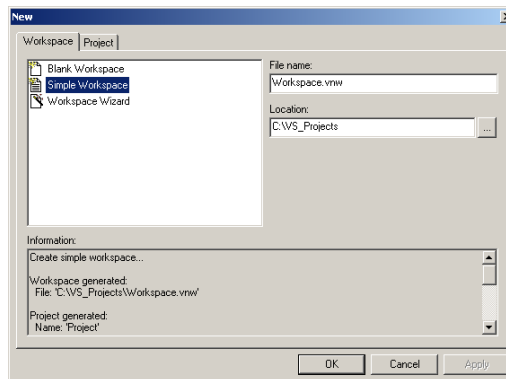


Figure 5: Navigator New dialog box

- 3 Under the **Workspace** tab, select **Simple Workspace**.

In the **File name** field and **Location** field you can specify file name and directory of the workspace file.

- 4 Click **OK**. The visualSTATE Designer application will be launched with a visualSTATE Project, System and topstate. See *Figure 6*, page 13.

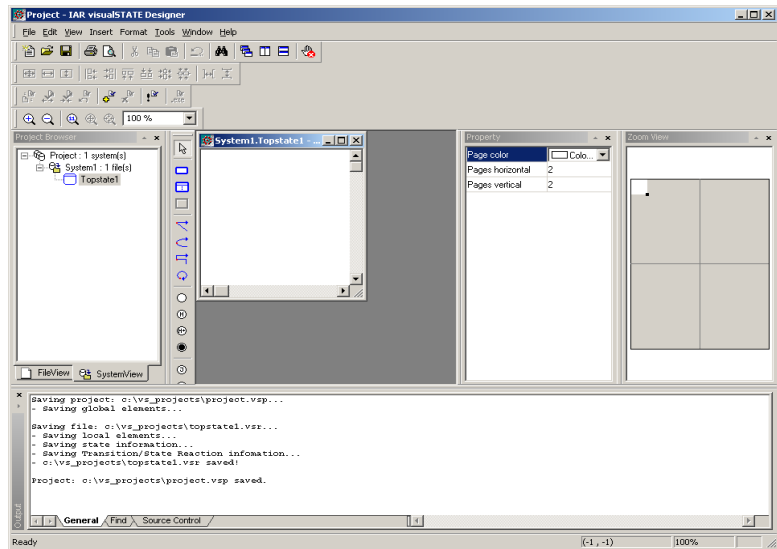


Figure 6: Designer application with newly created Project

Now you can start drawing statecharts for your visualSTATE model in the statechart diagram window of the Designer. See *Designing statechart diagrams*, page 57. When you have completed your statechart diagrams, save the Project in the Designer (choose File>Save Project).

- 5 Return to the Navigator. A reload message may be displayed. See *Figure 7*, page 14.

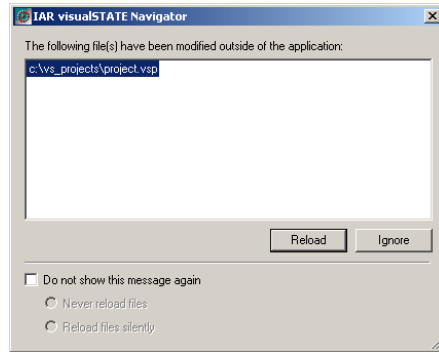


Figure 7: Navigator reload message

Click **Reload** to update the Project in the Navigator workspace. For information about reload of files in the Navigator, see *Reloading files in the Navigator*, page 32.

- 6 The workspace has now been set up with one Project, one System and one topstate, and default options for code generation, verification, and documentation of the visualSTATE Project. You can change these options by choosing Project>Options on the menu. See *Setting Verificator, Coder and Documenter options*, page 29.

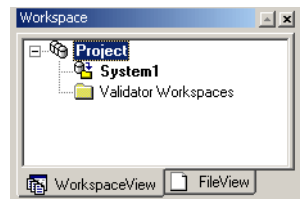


Figure 8: Workspace created in the Navigator

The workspace also contains a folder for Validator workspaces which are used for testing. See *Part 5: Functional testing*, page 147.

For creating a workspace with more than one System and Project, see *Creating and saving a workspace*, page 22.

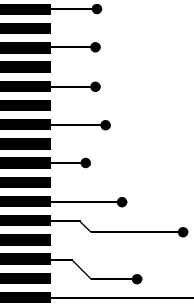
- 7 On the Navigator menu, choose File>Save Workspace.

Part 2: Project management

This part of the visualSTATE[®] User Guide includes the following chapters:

- Graphical environment
- Handling visualSTATE Projects, Systems and files
- Source code control
- Custom commands.





Graphical environment

For managing your visualSTATE Projects, you use the visualSTATE Navigator. This chapter describes the graphical environment of the Navigator, including toolbars. It also describes how you can customize the Navigator.

General

The graphical environment of the Navigator consists of a number of windows with context-sensitive pop-up menus, an integrated browser (similar to Microsoft Internet Explorer), menus and toolbars. *Figure 9*, page 17 shows the Navigator environment with a workspace loaded.

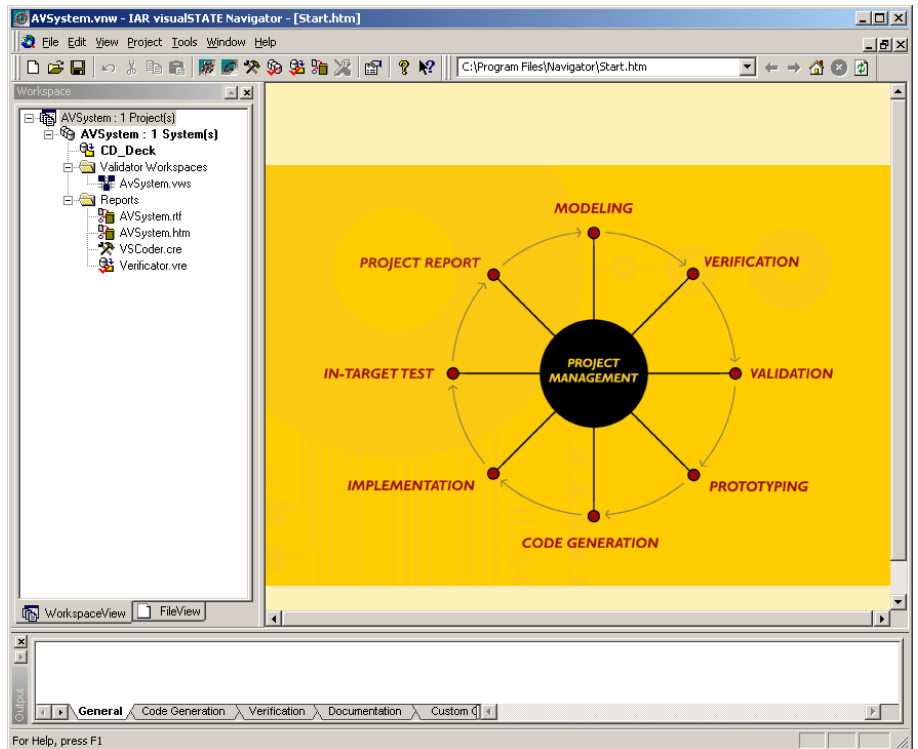


Figure 9: Navigator, with workspace loaded

Navigator windows

The Navigator has the following windows (see *Figure 9*, page 17):

- Workspace browser (opened via the View menu).
- Output window (opened via the View menu).
- HTML viewer.
- Properties window (opened via the View menu). See *Figure 10*, page 19.

WORKSPACE BROWSER

The left window of the Navigator shown in *Figure 9*, page 17 contains a workspace browser where you can see the structure of the loaded visualSTATE workspace. The browser has the following views:

- A file view which shows the file structure of the workspace file, with visualSTATE Project files, Statechart files and System folders.
- A workspace view which shows the model structure of the visualSTATE Projects in the workspace. This view also shows Project-related objects such as Validator workspaces and custom commands.

For a detailed description of the Navigator workspace, see *The workspace*, page 21.

OUTPUT WINDOW

This window displays information about the workspace loaded. The tabbed pages contain general information from the Verificator, Coder and Documenter when these tools have been activated.

HTML VIEWER

The Navigator HTML viewer is a window with an integrated Internet browser (the window in the right part of *Figure 9*, page 17).

On start-up of the Navigator, Navigator start page is displayed in an HTML viewer where you can activate the other visualSTATE modules.

Verification results and generated visualSTATE Project reports are also displayed in HTML viewers.

You can browse for other HTML pages by means of the Internet browser toolbar (see *Internet browser toolbar*, page 20). For each new HTML page, a new HTML viewer is opened.

To change between HTML viewers, use the commands on the Window menu.

PROPERTIES WINDOW

This window shows information about the currently active item in the workspace. See *Figure 10*, page 19.

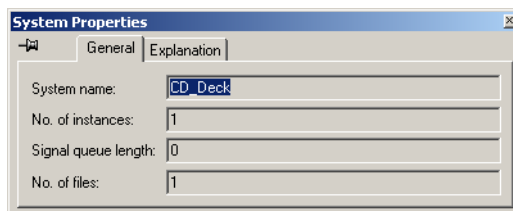



Figure 10: Navigator Properties window

To make the window remain on the screen, click .

Navigator toolbars

The most frequently used menu commands are available as toolbar buttons with tooltips. The following toolbars are available:

- Standard toolbar
- Internet browser toolbar.

If the toolbars are not visible, you can display them via the View menu.

A detailed description of the Navigator menu commands is found in *Navigator menu commands*, page 337).

STANDARD TOOLBAR

Figure 11, page 19 shows the Navigator Standard toolbar. The buttons on this toolbar correspond to the commands on the File, Edit, Project and Help menus.



Figure 11: Navigator Standard toolbar

INTERNET BROWSER TOOLBAR

Figure 12, page 20 shows the Navigator Internet browser toolbar. The buttons on this toolbar are used for searching on the Web, and correspond to the browse commands found on the View menu.

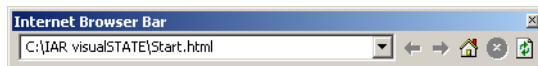


Figure 12: Navigator Internet browser toolbar

Customizing the Navigator

The Navigator can be configured to match your preferences with regard to HTML page shown at start up, location of user documentation files, display of warnings etc.

- 1 Launch the Navigator, and choose Tools>Settings. A dialog box is displayed. See Figure 13, page 20.

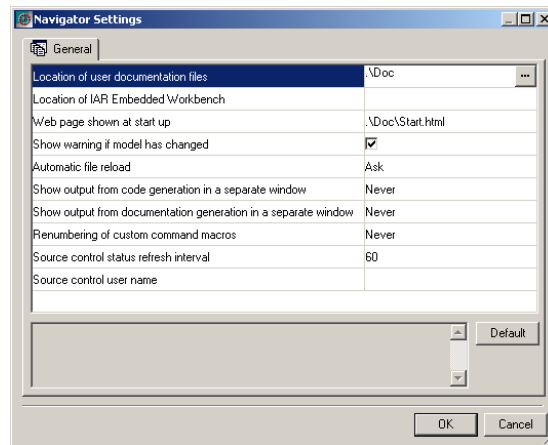


Figure 13: Navigator Settings dialog box

- 2 Click an option and type the appropriate values, or click the buttons that are shown when you click an option, and select values.

For a detailed description of the options, activate the online help by right-clicking an option, or pressing SHIFT+F1.

Handling visualSTATE Projects, Systems and files

This chapter describes the handling of visualSTATE Projects, Systems and Files by means of the Navigator workspace. It describes

- The workspace
- Creating and saving a workspace
- Opening a workspace
- Creating a new Project in a workspace
- Adding an existing Project to a workspace
- Removing a Project from a workspace
- Setting a Project or System as active
- Setting Verificator, Coder and Documenter options
- Reloading files in the Navigator
- Digital signature
- Handling Projects from previous visualSTATE versions
- Closing the Navigator.

The workspace

You set up your visualSTATE Project in a workspace using the Navigator. The workspace is a file for organizing and handling a collection of visualSTATE Projects, Systems and Statechart files that are grouped together logically. The workspace (extension `vnw`) contains links to visualSTATE Projects, Systems and various types of files.

When you have created a workspace (see *Creating and saving a workspace*, page 22), you can start working on your visualSTATE Projects by means of the other visualSTATE applications which you launch via the Navigator Project menu.

In the Navigator workspace you can set options for verification, code generation, and visualSTATE Project reports (see *Setting Verificator, Coder and Documenter options*, page 29). The workspace can also be used for setting up user-defined commands (see *Custom commands*, page 41).

The Navigator workspace should not be confused with the Validator workspace which is a workspace used for testing (see *The Validator workspace*, page 150).

PROJECTS IN WORKSPACE

You can have several visualSTATE Projects in the same workspace and different workspaces can contain the same Projects. A workspace file only contains one workspace.

A visualSTATE Project contains one or more visualSTATE Systems containing one or more visualSTATE Statechart files (extension `vst`). In the Navigator workspace browser, Systems are shown as folders containing `vst` files.

For a detailed description of visualSTATE Projects, Systems and Statechart files, refer to *IAR visualSTATE Reference Guide*.

Creating and saving a workspace

For creating a simple workspace, you can use the wizard described in *Setting up a visualSTATE Project*, page 12. If you want to be able to customize your workspace with regard to number of Projects and Systems, you can use the following methods:

- Creating a workspace using workspace wizard. This will launch the Designer.
- Creating a blank workspace. This is for example suitable when you want to create a new collection of Projects in a workspace.

CREATING A WORKSPACE USING WORKSPACE WIZARD

- 1 On the Navigator menu, choose File>New. A New dialog box is displayed. See *Figure 5*, page 12.
- 2 Under the Workspace tab, select **Workspace Wizard**.

In the File name field and Location field you can specify file name and directory of your workspace file.

- 3 A System(s) dialog box is displayed. See *Figure 14*, page 23.

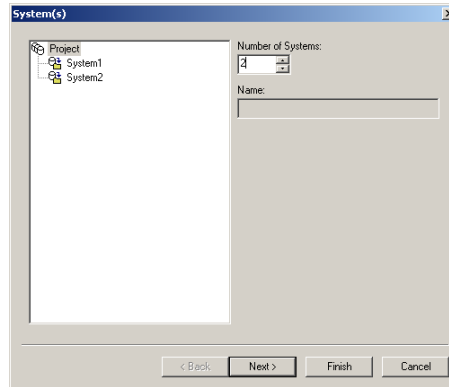


Figure 14: System(s) dialog box, Navigator

- 4 In the tree browser, select Project, and select number of visualSTATE Systems in the Number of Systems field.

You can change the name of a System by selecting it in the tree browser and typing a System name in the Name field.

- 5 Click *Next*. A Topstate(s) dialog box is displayed. See *Figure 15*, page 23.

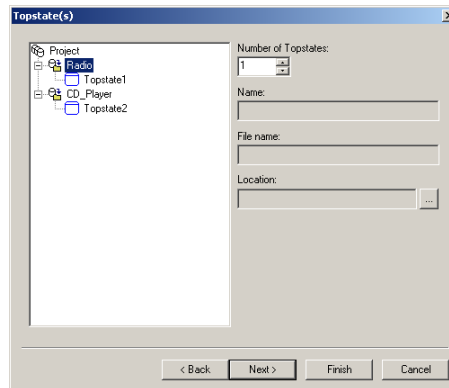


Figure 15: Topstate(s) dialog box, Navigator

- 6 In the tree browser, select a System. In the Number of Topstates field, select the number of topstates to be contained in the System.

You can change the topstate name, and name (extension `vsr`) and location of the topstate file.

- 7 Click **Next**. A Topstate Region(s) dialog box is displayed. See *Figure 16*, page 24.

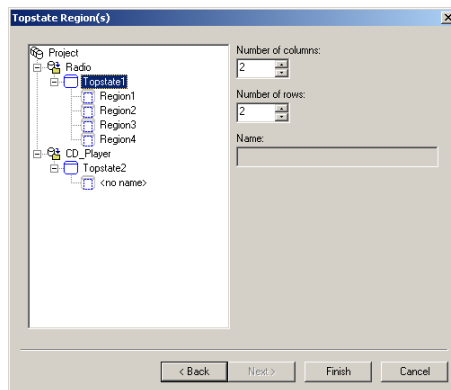


Figure 16: Topstate Region(s) dialog box, Navigator

- 8 In the tree browser, select a topstate. In the Number of columns field and Number of rows field, specify the number of regions to be contained in the topstate (horizontal and vertical distribution in Designer statechart diagram).

If a topstate has more than one region, each of its regions can be named using the Name field

- 9 Click **Finish**. A summary page is displayed informing you of the choices you have made.
- 10 Click **OK**. The visualSTATE Designer application will be launched with a Project containing the Systems and topstates you have specified in the wizard.

Now you can design your visualSTATE model in the statechart diagram window of the Designer. See *Designing statechart diagrams*, page 57. When you have completed your statechart diagrams, save the Project in the Designer (choose File>Save Project).

- 11 Return to the Navigator. Click **Reload** if the Reload message is displayed, to update the Project in the workspace (see *Figure 7*, page 14).

- 12 On the Navigator menu, choose File>Save Workspace.

The workspace will be saved. See example of workspace in *Figure 18*, page 27.

CREATING A BLANK WORKSPACE

- 1 On the Navigator menu, choose File>New.

- 2 Under the Workspace tab, select **Blank Workspace**.

In the File name field and Location field you can specify file name and directory of your workspace file.

- 3 Click **OK**. A blank workspace is created.

Now you can import existing Projects into the workspace (see *Adding an existing Project to a workspace*, page 27), or create a new Project with Systems in it, as described in *Creating a new Project in a workspace*, page 25.

Opening a workspace

You open a workspace as follows:

- 1 On the Navigator menu, choose File>Open Workspace.... An Open dialog box is displayed.
- 2 Specify file name, and/or browse for the directory where the file is located. Click **Open**.

The workspace file will be opened in the Navigator.

You can also open a workspace from the list of most recently used files by choosing File>number.

Creating a new Project in a workspace

To create a new Project in a workspace:

- 1 In the Navigator, open your workspace.
- 2 On the Navigator menu, choose File>New.

- 3 In the New dialog box displayed, click the Project tab. See *Figure 17*, page 26.

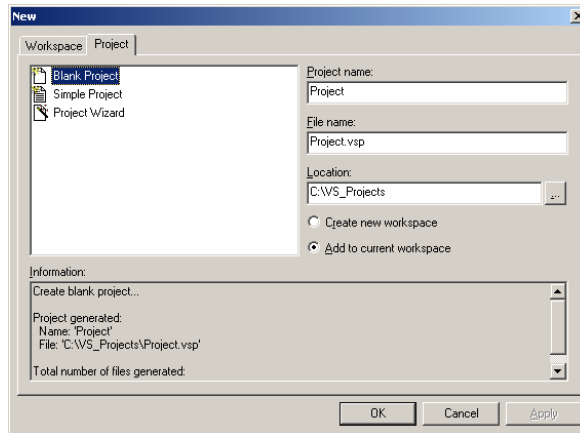


Figure 17: New dialog box, Project tab (Navigator)

- 4 Select one of the following:

To create a Project without Systems, select **Blank Project**. See *Creating a blank Project in a workspace*, page 26.

To create a simple Project with one System and one topstate, select **Simple Project**. See *Creating a simple Project in a workspace*, page 27.

To create a customized Project, select **Project Wizard**. See *Creating a Project using Project wizard*, page 27.

CREATING A BLANK PROJECT IN A WORKSPACE

- 1 When you have selected **Blank Project** under the Project tab of the New dialog box (see *Figure 17*, page 26), specify Project name, Project file name (extension `vsp`), and location of Project file.
- 2 Select **Add to current workspace**, and click **OK**. The Designer will be launched with a blank Project where you can create Systems and topstates (see *Creating Systems and Statechart files in a blank Project*, page 99).

Note: Selecting **Create new workspace** here will generate a workspace file with a file name composed of the Project name and the extension `vnw`. The workspace file will be located in the same directory as the `vsp` file. The Project will be inserted in the newly created workspace and the Designer will be launched with the Project.

- 3 Return to the Navigator. Click **Reload** if the Reload message is displayed, to update the Project in the workspace (see *Figure 7*, page 14). The new Project is inserted in the workspace.

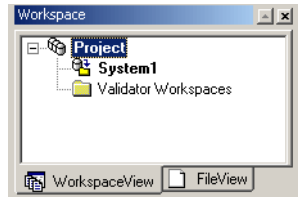


Figure 18: Workspace browser with Project

CREATING A SIMPLE PROJECT IN A WORKSPACE

- 1 When you have selected **Simple Project** under the Project tab of the New dialog box (see *Figure 17*, page 26), specify Project name, Project file name (extension `vsp`), and location of Project file.
- 2 Select **Add to current workspace**, and click **OK**. The Designer will be launched with a Project containing one System and one topstate.
- 3 Return to the Navigator. Click **Reload** if the Reload message is displayed, to update the Project in the workspace. The new Project is created in the workspace. See example in *Figure 18*, page 27.
- 4 On the Navigator menu, choose File>Save Workspace.

CREATING A PROJECT USING PROJECT WIZARD

- 1 When you have selected **Project Wizard** under the Project tab of the New dialog box (see *Figure 17*, page 26), specify Project name, Project file name (extension `vsp`), and location of Project file.
- 2 Select **Add to current workspace**, and click **OK**. The first page of the wizard is displayed. See *Figure 14*, page 23.
- 3 Perform *Step 4*, page 23, to *Step 12*, page 24.

Adding an existing Project to a workspace

visualSTATE Projects created during another session, or with the Designer, can be imported into a Navigator workspace as follows:

- 1 In the Navigator, open the workspace into which you want to import a Project.

- 2 On the menu, choose File>Insert Project. A dialog box is displayed, see *Figure 19*, page 28.

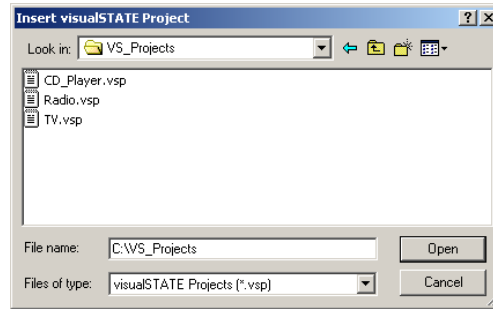


Figure 19: Insert visualSTATE Project dialog box

For information on how to create Systems and Statechart files in a Project, see *Creating and saving Projects, Systems, and files in the Designer*, page 97.

Removing a Project from a workspace

To remove a Project from a workspace:

- 1 In the Navigator, open the workspace from which you want to remove a Project.
- 2 In the workspace browser, click the File view tab, and select the Project to remove.
- 3 Open the pop-up menu and choose **Remove**.

Setting a Project or System as active

You can set a Project or System as active. This means that all operations that you perform via the main menu will apply only to that Project or System. For example Project>Verify System will start verification of the active System in the active Project.

To set a Project or System as active:

- 1 Open your workspace in the Navigator.

- 2 In workspace browser, select the System or Project to set as active. Open the pop-up menu and choose **Set as Active...**. See *Figure 20*, page 29 where a Project has been selected.

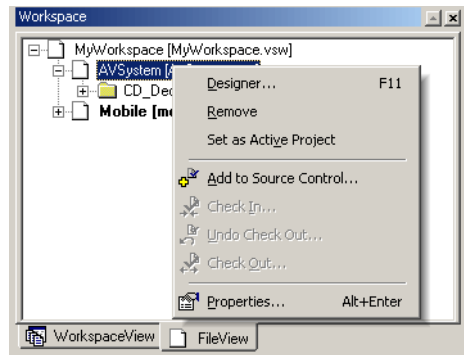


Figure 20: Setting a visualSTATE Project as active

The item that has been set as active is shown in bold in the workspace browser.

If you want to apply operations to Projects or Systems not set as active, you can select the item in the Workspace view of the browser and use the commands on the pop-up.

Setting Verificator, Coder and Documenter options

For verification, code generation, and documentation of visualSTATE Projects you can specify a variety of options, as follows:

- 1 Launch the Navigator, and open your workspace file.
- 2 To set Verificator options, choose Project>Options>Verification.... (Verificator options are described in *Part 4: Formal testing*, page 107).

To set Coder options, choose Project>Options>Code Generation.... (Coder options are described in *Part 7: Code generation*, page 231).

To set Documenter options, choose Project>Options>Documentation.... (Documenter options are described in *Part 8: Documenting visualSTATE Projects*, page 255).

An Options dialog box is displayed. See *Figure 21*, page 30.

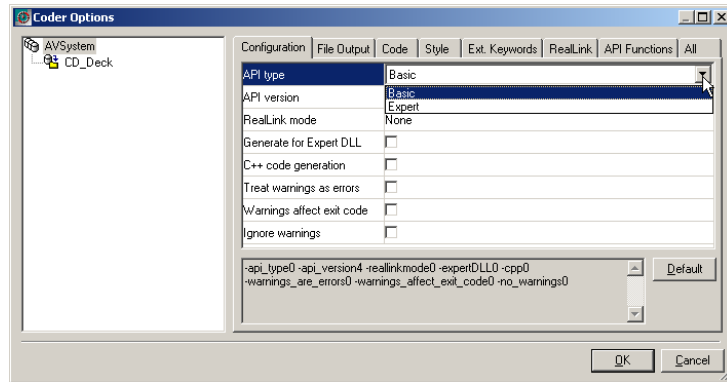


Figure 21: Coder Project Options dialog box, Configuration tab

Here the use of the Coder Options dialog box will be explained. The Verificator and Documenter options dialog boxes are used in the same way.

For a detailed description of the options, see *Verificator command line options*, page 371, *Coder options*, page 375, and *Setting up a visualSTATE Project report*, page 261.

- 3 In the Project browser to the left, select the visualSTATE Project or System for which to apply options.
- 4 Click the tab containing the category of options you want to set. To view all options available, click the All tab.
- 5 Click an option. There are several methods of setting values for options, depending on the option selected:
 - For some options there is a drop-down list box to the right of the option. Click in the list box and select the appropriate value. See *Figure 21*, page 30.
 - Other options have check boxes. Click the option check box to select or deselect the option. See *Figure 21*, page 30.
 - For some options, you can type the value in the field to the right of the option.
 - Some options have buttons that are displayed when you click the option:



By clicking this button you can browse for files to use.



Clicking this button will display a pop-up menu.

The selected values are shown as command line options in the pane below the option list. See example in *Figure 21*, page 30.

Dimmed options cannot be changed. The reason is that not all combinations of options are possible. Thus the values selected for one option may exclude choices for other options. This is described in the online help for the option (see *Online help*, page 31).

- You can restore the options to their default values by clicking the **Default** button.

ONLINE HELP

You activate the online help for Verificator, Coder, and Documenter options by right-clicking the option, or pressing SHIFT+F1. See *Figure 22*, page 31.

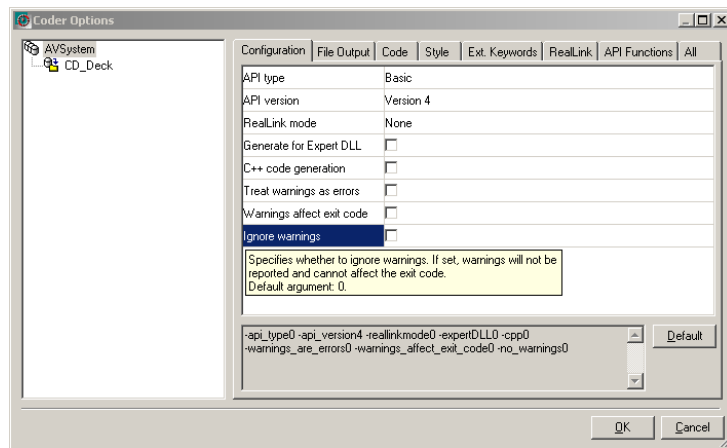


Figure 22: Display of online help

Reloading files in the Navigator

By default, you will receive a reload message in the Navigator when the Project files or Statechart files (`vsp` and `vsr` files) contained in the current workspace have been modified outside the Navigator. See *Figure 23*, page 32.

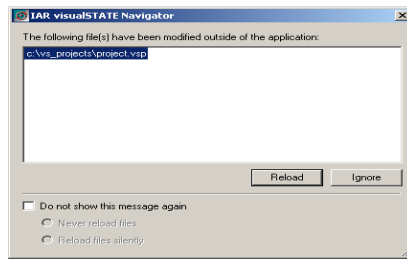


Figure 23: Navigator reload message

If you click **Reload**, the information about all modified Projects and Systems in the workspace browser will be updated. Note that only the *graphical information* is reloaded, not the information in the workspace file about links to the modified Projects and Systems.

To change the way the reload message is displayed, do this:

- I In the Navigator, choose Tools>Settings. A dialog box is displayed. See *Figure 24*, page 32.

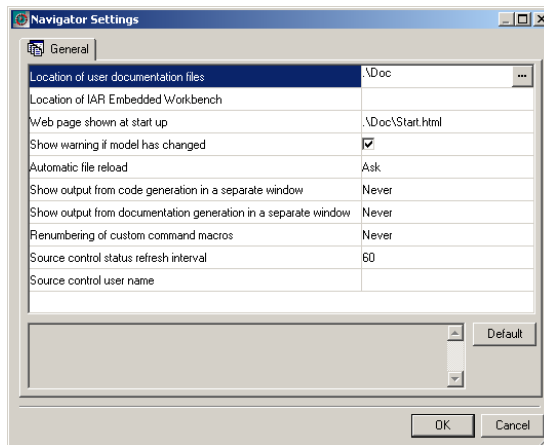


Figure 24: Navigator Settings dialog box

- 2 Select *Automatic file reload*, and click the arrow button displayed to the right of the option. Select one of the following:

Ask	This is the default setting. If one or more files are changed, the reload message shown in <i>Figure 23</i> , page 32 will be displayed, and you can choose to refresh the view of the files in the workspace by selecting <i>Reload</i> . If you choose not to reload, no graphical update will be made in the workspace (corresponds to the setting <i>Never</i> , see below).
Always	All relevant <code>vsp</code> and <code>vsr</code> files are reloaded automatically when they have been changed outside the Navigator. The graphical information about Projects and Systems in the workspace is updated, just as reload information is written to the output window (General tab).
Never	Changes in <code>vsp</code> and <code>vsr</code> files are ignored in the graphical representation of Projects and Systems in the Navigator workspace browser. The only way to update the graphical information is to close and reopen the workspace. This setting is not recommended.

Digital signature

A digital signature is associated with each visualSTATE Project. The purpose of the digital signature is to track consistency between the files generated by the various visualSTATE programs, and to track changes from version to version of a visualSTATE Project.

The digital signature is a string value based on a visualSTATE Project file and its associated Statechart files. The digital signature value does not depend on all parts of a visualSTATE Project; it only depends on the logical parts of the visualSTATE Project. Other parts of a visualSTATE Project, for example explanations to various visualSTATE elements, do not have any influence on the digital signature. Whenever a visualSTATE Project part is changed on which the digital signature depends, for example the renaming of an event, the digital signature is changed.

The digital signature is used in the following cases:

- visualSTATE Project code-generated via the Navigator
- Generated files included in Documenter report.
- Runtime application.

VISUALSTATE PROJECT CODE-GENERATED VIA THE NAVIGATOR

When a visualSTATE Project is code-generated via the Navigator, its digital signature is saved for later use. When you later choose to start verification of the Project via the Navigator, the Navigator will issue a notification if the saved digital signature from the last code generation differs from the current digital signature, and you have chosen to be notified of model changes. Thus a notification will be issued in the following cases:

- when an attempt is made to verify a visualSTATE Project or visualSTATE System for which you have not generated code.
- the Coder-generated files are not consistent with the model due to changes in the model.

See *Figure 25*, page 34.

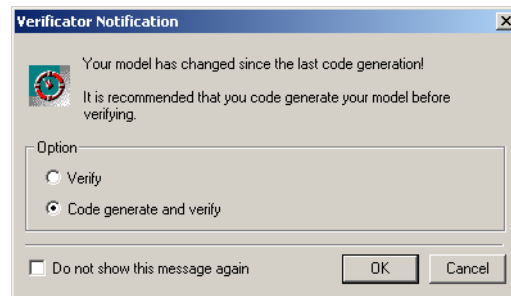


Figure 25: Verificator notification

To set up notification on model change, do the following:

- 1 In the Navigator, choose Tools>Settings. The Navigator Settings dialog box is displayed. See *Figure 13*, page 20.
- 2 Select *Show notification if model has changed*.

GENERATED FILES INCLUDED IN DOCUMENTER REPORT

A digital signature is included in all files generated by the Validator, Coder and Verificator that can be included in a Documenter-generated report. By default Documenter will only include files with a correct digital signature. The default behavior can be changed via Documenter options. See *Specifying visualSTATE files to be used as input for Project report*, page 263.

For information about creating a Documenter report, see *Part 8: Documenting visualSTATE Projects*, page 255.

RUNTIME APPLICATION

The Coder stores the digital signature in one of the Coder-generated files. At runtime, the digital signature can be retrieved from the embedded environment. By searching through a file system it is possible to find the visualSTATE Project associated with this digital signature. The digital signature is stored in the `vsp` file whenever the visualSTATE Project is saved.

Handling Projects from previous visualSTATE versions

When you open a visualSTATE Project of a previous visualSTATE version in version 5, you must convert the Project to version 5. See *Figure 26*, page 35.

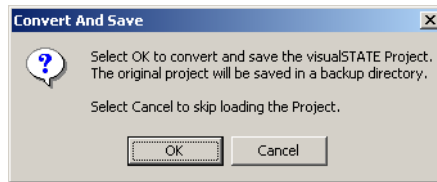


Figure 26: Conversion of Project from previous visualSTATE version

If you select **OK**, the Project will be converted and you can edit it in the Designer. Converted files are located in the original Project directory. A copy of the original files are located in an automatically created directory named `Backup` below the Project directory.

If you select **Cancel** in the dialog box shown in *Figure 26*, page 35, the Project will not be converted and the load will be terminated.

For additional information about conversion of visualSTATE Projects, see *Appendix D: Handling visualSTATE files from previous versions*, page 435.

Closing the Navigator

You close the Navigator by choosing `File>Exit`.

When the Navigator is closed, all running instances of the Designer, Validator, and IAR Embedded Workbench® that were opened by the Navigator will be closed too.

Source code control

The files created in the visualSTATE Navigator and Designer can be added to a source code control system. You can either use the built-in visualSTATE MultiUser Management for source code control, or you can use any system that supports Microsoft Common Source Code Control (Microsoft SCC API), for example Microsoft Visual SourceSafe.

This chapter describes how to you can use the Navigator and Designer for source code control for your visualSTATE files.

Supported visualSTATE file types

The following visualSTATE file types can be added to a source code control system:

Navigator-created files: `vnw`, `vtg`

Designer-created files: `vsp`, `vsr`

Validator-created files: `vws`

Files created by third-party product: `oil`

Using source code control

You add files to source code control using the Navigator or the Designer:

- To add `vsp` and `vsr` files, you use the Designer.
- To add `vnw`, `vtg`, `vws`, and `oil` files, you use the Navigator.

The following procedure describes how to add Designer-created files to source code control. The same procedure applies to files that are handled by the Navigator.

To add Designer-created file to source code control:

- 1 Launch the Designer and open you visualSTATE Project.
- 2 In the Project browser, select the file to add to source code control. Open the pop-up menu and choose *Add to Source Control*.
- 3 In the dialog box displayed, select the source code control system to apply.

The file will be added to the source code control system and shown with gray icons in the browser.

ACCESSING FILES UNDER SOURCE CODE CONTROL

To edit a supported visualSTATE file that has been added to a source code control system, it must be checked out to the user who wants to edit it. Only one user at a time can edit a file.

You check out visualSTATE files using the Navigator or Designer, or a third-party source code control system. The following procedure describes how to check out Designer created files. The same procedure applies to files that are handled by the Navigator (see *Using source code control*, page 37).

You check out a file using the Designer as follows:

- 1 Open your Project file. In the Project browser, select the files to check out. Open the pop-up menu and choose Source Control>Check Out.

The checked-out files will be shown with a red check mark in the browser.

You can also use a third-party source code control system to check out the files. Again, they will be marked with red check marks in the Designer and Navigator browsers.

- 2 When you have completed editing your files, you check them in by selecting Source Control>Check In....

Retrieving latest copy of file

To retrieve the latest copy of a file under source code control, choose File>Source Control>Get Latest Version.... Note that this operation will typically not check out the file and make it editable (depends on source code control system).

SOURCE CODE CONTROL STATUS OF FILES

The source code control status of files are shown in the Navigator and Designer browsers by icons. If a file is checked in, it is shown with gray icons in the browser. If a file is checked out and can be edited by the current user, it is shown with red check marks.

To update the source code control status of the files, choose File>Source Control>Refresh Status on the menu.

Update frequency of source code control status

You can set up how often the source code control status in the Navigator workspace browser should be updated, as follows:

- 1 On the Navigator menu, choose Tools>Settings.

- 2 Click the **Source control status refresh interval** option and type number of seconds in the field to the right. See *Figure 27*, page 39.

Note: If you enter the value zero in this field, regular update of the source code control status will be disabled.

User name for source code control system

Some source code control systems require a user name for login which you specify in the Navigator as follows:

- 1 On the menu, choose Tools>Settings.
- 2 Click the **Source control user name** option and type user name in the field to the right. See *Figure 27*, page 39.

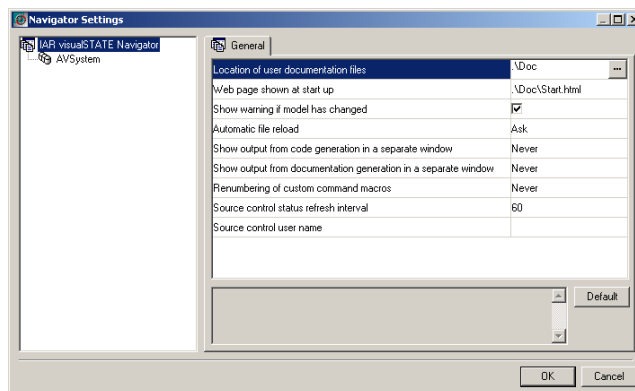


Figure 27: Navigator Settings dialog box

Note: If you leave **Source control user name** blank, the Windows user name will be used for source code control login.

User name for source code control system

Custom commands

This chapter explains the following:

- What is a custom command?
- Creating custom commands
- Activating custom commands
- Editing, renaming, and deleting custom commands
- Renumbering of custom command macros.

What is a custom command?

A custom command is a user-specified command for performing a specific task, for example compilation of an entire visualSTATE Project.

You can set up one or several custom commands for each Project in a Navigator workspace, and for the workspace itself.

Note: Custom commands are workspace-specific, that is, they apply only to the workspace where they have been created and its Projects.

Creating custom commands

You set up a custom command as follows:

- I Launch the Navigator, and open your workspace file.

- 2 On the menu, choose Tools>Custom Commands. A Custom Commands dialog box is displayed. See *Figure 28*, page 42.

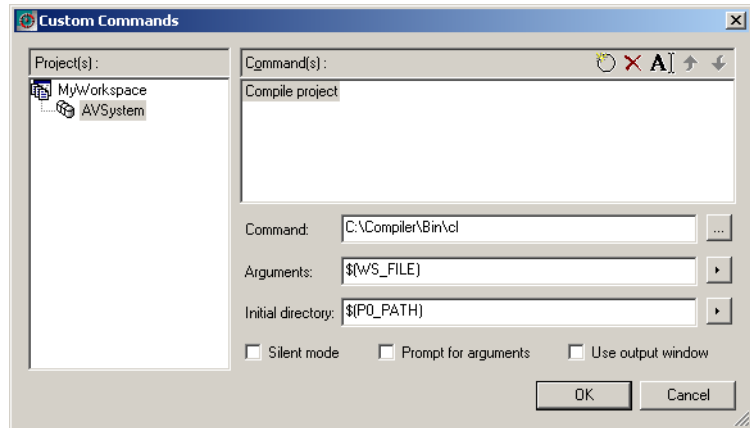


Figure 28: Custom commands dialog box (Navigator)

Here you specify which command to execute.

- 3 The Project(s) tree in the left pane shows the loaded visualSTATE Projects. The root item is the visualSTATE workspace name. Each node in the tree represents a visualSTATE Project.


To create a workspace-specific custom command, select the visualSTATE workspace.


To create a Project-specific custom command, select the visualSTATE Project.

Note: Workspace-specific custom commands have access to the workspace and all Projects and Systems contained in it. Project-specific custom commands only have access to the Project where they are defined and the Systems contained in it.

- 4 The Command(s) section shows the custom commands created for the workspace or selected Project.

On the Command(s) toolbar, click the  button to create a new command.

- 5 In the Command field, click the  button to browse for path and name of the program to be executed.

- 6 In the Arguments field, you specify the arguments to be used by the custom command. Either type the arguments, or click the  button to display a pop-up menu of arguments. See *Figure 29*, page 43.

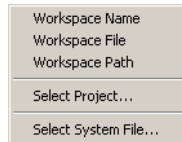


Figure 29: Custom commands, arguments pop-up menu (Navigator)

If you choose *Select Project...*, or *Select System File...*, a Select dialog box is displayed. See *Figure 30*, page 43.

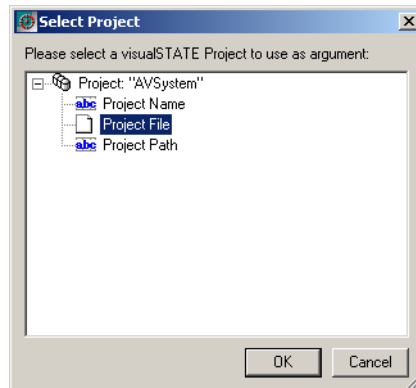


Figure 30: Custom commands, Select Project dialog box (Navigator)

Select the appropriate item to use as argument.

Example

Selecting `Project File` for the first Project in the workspace will insert the macro `$(P0_FILE)` in the Arguments field of the Custom Commands dialog box (see *Figure 28*, page 42). When the custom command is activated, `$(P0_FILE)` is expanded to the name of the first Project file in the workspace, for example `Project.vsp`.

See also *Renumbering of custom command macros*, page 45.

- 7 In the Initial directory field of the Custom Command dialog box, click the right arrow button to select the directory to change to during execution of the custom command.

- 8 Select **Silent mode** if you do not want any windows to be displayed during execution of the custom command.

Note: Use this function with caution. Any windows requesting user interaction during the execution of the command will not be shown.
- 9 Select **Prompt for arguments** to be prompted for arguments during execution of the custom command.
- 10 Select **Use Output window** to have any console output displayed on the Custom Command tab page of the Navigator output window (such as the `echo` DOS command and output from most compilers called from the command line).
- 11 To change the order in which custom commands appear in the workspace browser, click the up and down arrows on the Commands toolbar of the Custom Commands dialog box.
- 12 When you have created a custom command, it is added to the Project Custom Commands folder in the workspace browser. *Figure 31*, page 44 shows a Navigator workspace with a custom command named `Compile project`.

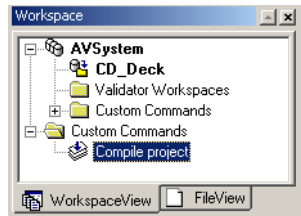


Figure 31: Navigator workspace with custom command

- 13 Save the workspace.



Activating custom commands

You activate a custom command by double-clicking it in the Navigator workspace browser. See *Figure 31*, page 44.

Editing, renaming, and deleting custom commands

You edit, rename, and delete custom commands as follows:

- 1 On the menu, choose **Tools>Custom Commands**. The Custom Commands dialog box is displayed. See *Figure 28*, page 42.

- 2 In the Project(s) tree, select a command and edit, rename, or delete it using the  and  buttons on the toolbar.

Renumbering of custom command macros

You can set up whether the numbering of custom command macros for arguments and initial directory should be changed when a Project or System is created or deleted in a Navigator workspace.

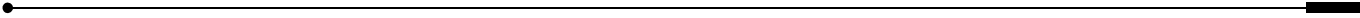
To set custom commands renumbering:

- 1 On the Navigator menu, choose Tools>Settings.
- 2 On the General tab, select *Renumbering of custom commands* and click the drop-down list. Select the appropriate option: *Never, Ask, Always*.

Part 3: Modeling

This part of the visualSTATE[®] User Guide includes the following chapters:

- Graphical environment
- Getting started
- States
- Transitions
- Elements
- Handling Projects, Systems, and files for modeling.





Graphical environment

For designing your visualSTATE models, you use the visualSTATE Designer. This chapter describes the graphical environment of the Designer, including toolbars.

General

The Designer environment consists of a Project browser, menus, toolbars, and a number of windows with pop-up menus. *Figure 32, page 49* shows the Designer environment with a Project loaded.

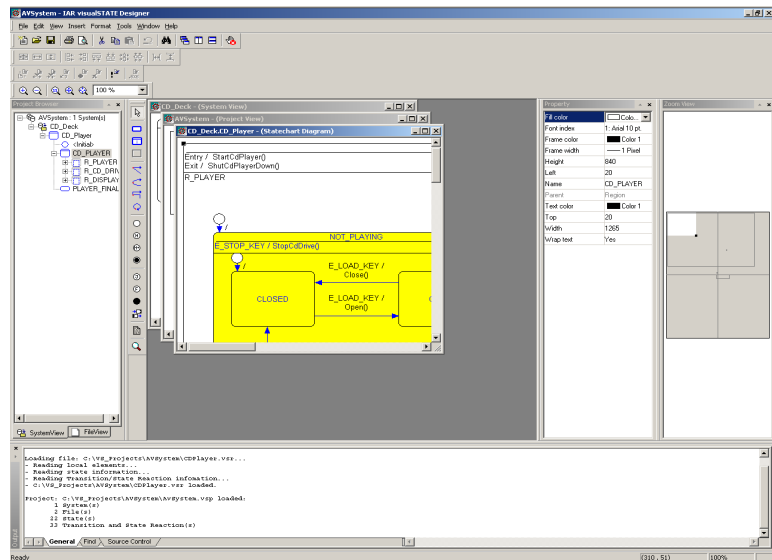


Figure 32: Designer environment with visualSTATE Project

Most windows and objects have pop-up menus which you activate by right-clicking the object. See example in *Figure 33*, page 50.

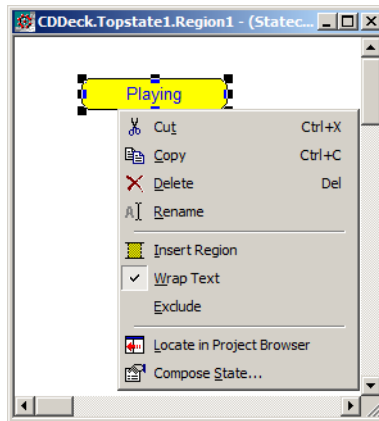


Figure 33: State pop-up menu

Designer windows

The Designer windows are opened via the View menu and are the following:

- Project browser window
- Diagram window
- Element browser window
- Property window
- Output window
- Zoom view.

PROJECT BROWSER WINDOW

This is the left-most pane of the Designer windows (see *Figure 32*, page 49). It shows the structure of the visualSTATE Project and is used for navigating through the Systems, topstates, regions, and states of a visualSTATE Project. For example, you can display the individual states in the statechart diagrams by double-clicking their names in the Project browser.

The Project browser has two views which show the structure of the Project on file level and statechart level respectively. Click the tabs to change between the two views.

DIAGRAM WINDOW

This window is used for designing the logic of the visualSTATE System by means of statechart diagrams. The diagram window has the following views:

- Project view
- System view
- Statechart diagram view.

See *Figure 34*, page 51.

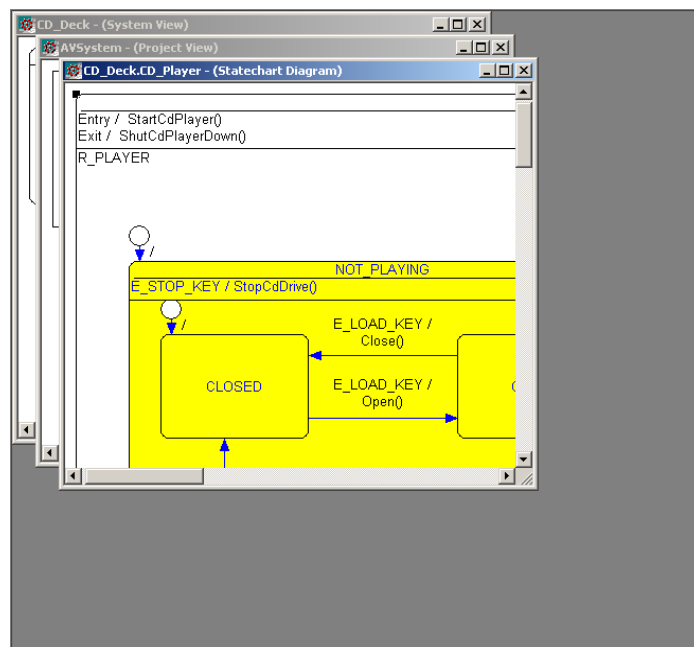


Figure 34: Designer diagram window

To change between the views, click the appropriate item in the Project Browser window. For example, to display the Project view of the diagram window, click the Project icon in the browser.

To view the details of a specific area or object in a diagram, click the item, and click the Zoom In toolbar button.

ELEMENT BROWSER WINDOW

This window is used for creating and browsing for the elements created for states and transitions, as described in *Elements*, page 89.

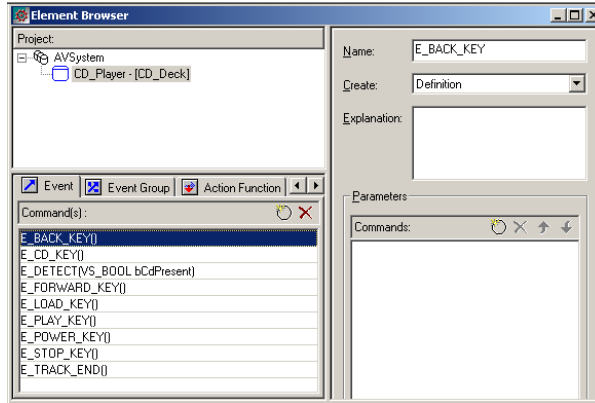


Figure 35: Designer element browser window

PROPERTY WINDOW

In this window you can specify the properties of objects in diagrams, for example font types for state names, colors of transitions etc. See *Figure 36*, page 52.

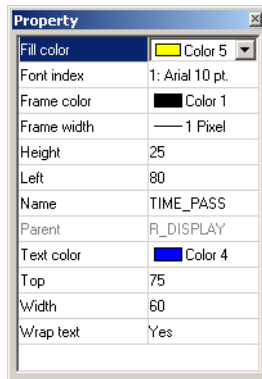


Figure 36: Designer property window

OUTPUT WINDOW

The output window is placed at the bottom of the screen and contains information about the loaded Project, result messages on element searches, and source code control information (see *Searching for an element*, page 95, and *Source code control*, page 37).

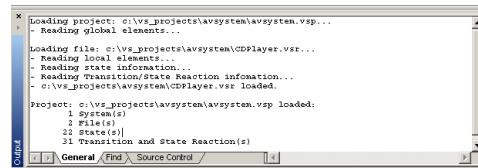


Figure 37: Designer output window

To change between the views, click the tabs, or press CTRL+TAB.

ZOOM VIEW

In this window you can see your entire drawing area and the position of your current diagram in the drawing area (see *Navigating in statechart diagrams*, page 60).

Designer toolbars

The most frequently used menu commands are available as toolbar buttons with tooltips. A detailed description of the Designer menu commands is found in *Designer menu commands*, page 345.

The following toolbars are available:

- Standard toolbar
- Diagram toolbar
- Size toolbar
- Source Control toolbar
- Zoom toolbar.

If the toolbars are not visible, you can display them via the View menu.

STANDARD TOOLBAR

Figure 38, page 53 shows the Designer Standard toolbar.



Figure 38: Designer Standard toolbar

The buttons on this toolbar correspond to the commands on the File, Edit, Tools and Window menus.

DIAGRAM TOOLBAR

Figure 39, page 54 shows the Diagram toolbar.



Figure 39: Designer Diagram toolbar

The buttons on this toolbar correspond to the commands on the Insert menu.

SIZE TOOLBAR

Figure 40, page 54 shows the Size toolbar.



Figure 40: Designer Size toolbar

The buttons on this toolbar correspond to the commands on the Format menu.

Note: Objects must be selected for the buttons to be available.

SOURCE CONTROL TOOLBAR

Figure 41, page 54 shows the Source Control toolbar.



Figure 41: Designer Source Control toolbar

The buttons on this toolbar correspond to the commands on the File>Source Control menu.

Note: File must have been added to source control system for the buttons to be available.

See also *Source code control*, page 37.

ZOOM TOOLBAR

Figure 42, page 55 shows the Zoom toolbar.



Figure 42: Designer Zoom toolbar

The buttons on this toolbar correspond to the zoom commands on the View menu.

Getting started

This chapter describes how to get started designing statechart diagrams in visualSTATE Designer, and gives a general introduction on how to use the Designer.

Before you can start designing statechart diagrams, a Project with a System and Statechart file must be created in a Navigator workspace (see *Setting up a visualSTATE Project*, page 12) or in the Designer (see *Handling Projects, Systems, and files for modeling*, page 97).

Shortcuts are listed in *Designer shortcuts*, page 341.

Designing statechart diagrams

To start designing a statechart diagram, you first create a workspace in the Navigator as described in *Setting up a visualSTATE Project*, page 12. The Designer will be launched with the visualSTATE Project loaded in the Navigator workspace (see *Figure 43*, page 57).

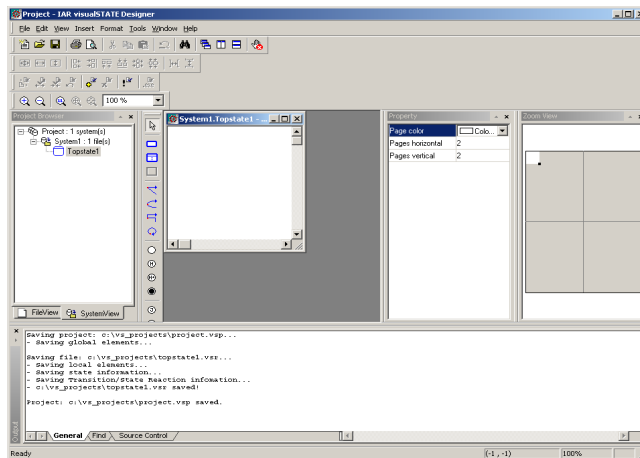



Figure 43: Designer with Project loaded

Now you can begin designing the actual statechart diagram which is drawn in the statechart diagram view.

DRAWING A SIMPLE STATE

To draw a simple state:

- 1 On the Diagram toolbar, click the Simple State button () , go to the statechart diagram view, and click on it. A default state will be created in the diagram.

To create a state with another size, click the Simple State button. Then go to the statechart diagram, press and hold the left mouse button while dragging a rectangle. Release the mouse button.
- 2 Deactivate the Simple State tool by right-clicking the mouse. The state you have drawn can be resized and moved as necessary by dragging it.

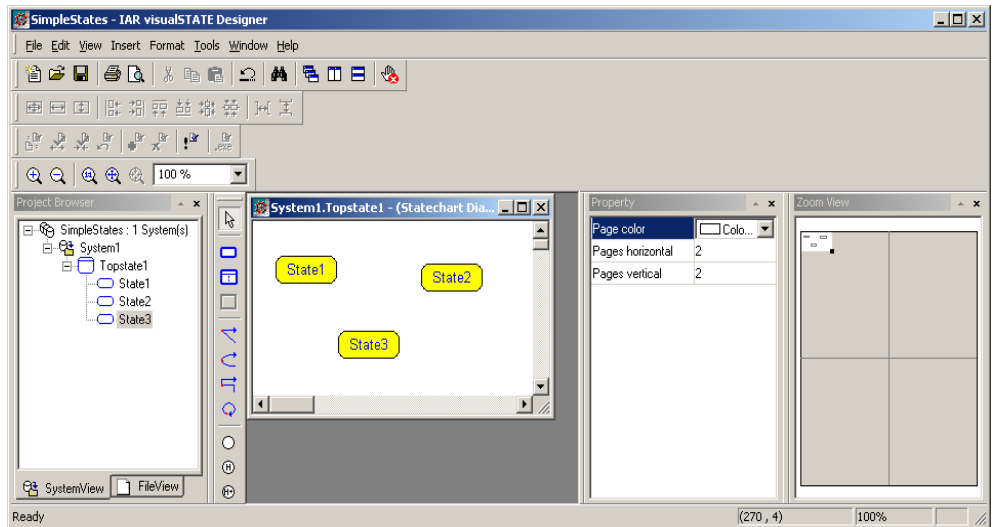


Figure 44: Newly drawn states


- 3 By default a state is given a name `State#`. To change the name, click on the default state name, and type a new name.

You can compose states with elements as described in *Composing states*, page 67. You can change color, frame width etc. using the property window (opened via `View>Property`).

When you have drawn a number of states, you can draw transitions between them.

DRAWING A TRANSITION BETWEEN TWO STATES

To draw a transition:

- 1 On the Diagram toolbar, click the Add Transition button ()
- 2 Move the cursor onto the source state in the statechart diagram view, and click the left mouse button. The frame of the source state is highlighted, and a hook point is displayed.
- 3 To start drawing the transition, release the left mouse button and move the cursor to the frame of destination state. The frame of the destination state is highlighted. Click the frame of the destination state. The transition line is drawn between the states.
- 4 Deactivate the transition tool by right-clicking the mouse.

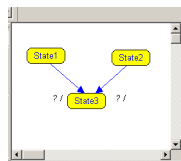


Figure 45: Examples of transitions

Transition description

The "? /" on the transition is the transition description. The "?" illustrates that the transition does not have a trigger. The "/" is a separator which divides the transition into a condition side and an action side. Composing a transition with a condition side and an action side is described in *Composing transitions*, page 83.

The transition description is locked to an anchor so that the description moves with the transition. You can set up location of the description on the transition (start, center, end), as follows:

- 1 Choose Tools>Customize... on the menu. A Customize dialog box is displayed.
- 2 In the tree in the left-hand pane of the dialog box, double-click on Transition, select transition, and specify anchor position in the right-hand pane.

Route points

Transitions have route points. You can choose to turn off route points by choosing Tools>Settings... on the menu. In the Settings dialog box displayed, click the Transition tab and deselect *Show route points*.

EDITING OBJECTS IN STATECHART DIAGRAMS


In the Designer you can rename Projects, Systems, topstates, regions, states, composite states, and substates, and change alias names and explanation notes, as follows:

- 1 In the diagram window, click the item to edit.
- 2 To rename an object, type a new name and press ENTER, or select the item in the Project browser, open the pop-up menu, select **Rename**, and type a new name.
- 3 To enter or change alias names and explanation notes, open the pop-up menu, and choose **Compose...** A Compose dialog box is displayed where you can enter and edit alias and explanation.

For renaming of elements, see *Creating and editing elements*, page 89.

STATECHART NOTES

You can insert notes anywhere in a statechart diagram as follows:

- 1 On the Designer Diagram toolbar, click the Note button ().
- 2 Click in the statechart diagram where you want to place the note. A rectangle is inserted.
- 3 Right-click to deactivate the Notes tool.
- 4 To write text in the note, click the frame of the note. It will be marked with black squares. Start typing. To insert a line break, press CTRL+ENTER.
- 5 Press ENTER to finish typing.

DELETING OBJECTS IN STATECHART DIAGRAMS

You can delete Systems, topstates, regions, states, composites states, substates, and transitions in the Designer as follows:

- 1 Open your visualSTATE Project.
- 2 Go to the appropriate view in the diagram window, select the object to delete, and press the DELETE button, or choose **Delete** from the pop-up menu.

Items displayed in the Project browser can be deleted as follows: Select the item in the Project browser, and choose **Delete** from the pop-up menu.

Navigating in statechart diagrams

To find out where in a statechart diagram objects are located, you use the zoom view. You can move to another area of the statechart diagram, by dragging the white square in

the zoom view where the gray area represents the entire statechart diagram area. See *Figure 46*, page 61 and *Figure 47*, page 62.

When you use the scroll bars of the statechart diagram view, it is reflected in the position of the white square in the zoom view.

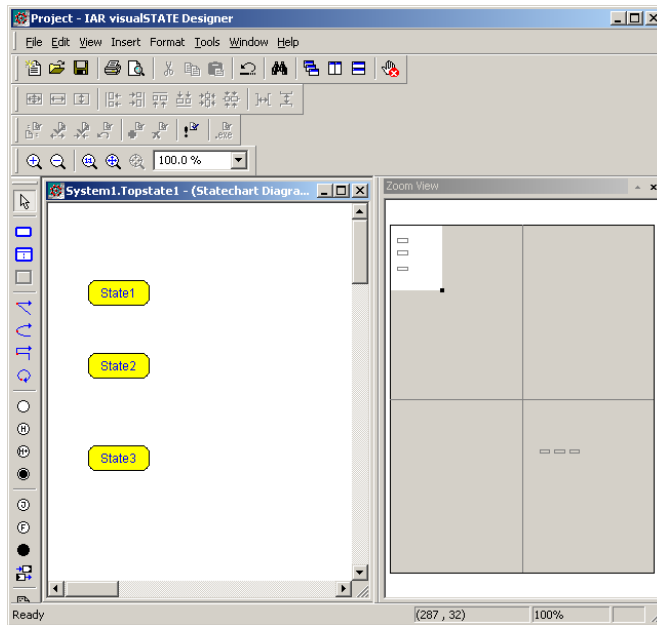


Figure 46: Designer zoom view, focus on upper left part of statechart diagram

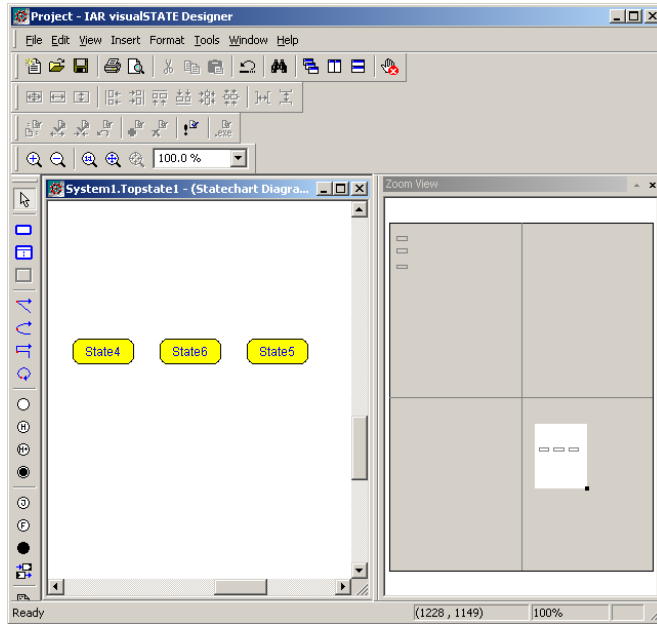


Figure 47: Designer zoom view, focus on lower right part of statechart diagram

To get detailed information about an object in the diagram window, move the cursor onto it to have tooltips displayed.

SELECTING A COLLECTION OF OBJECTS

You can select a collection of objects in the statechart diagram window as follows:

- I In the diagram, not on the objects to be selected, click the left mouse button, hold it down and draw a box (dotted line) around the objects. See *Figure 48*, page 62.

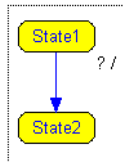


Figure 48: Objects selected

Or click the individual objects while holding down the SHIFT key.

- 2 Release the mouse button. The objects will be marked with square selection marks.


Now the objects can be changed. For example you can change the color of selected states using the property window, or align them using the Alignment toolbar (see *Resizing and positioning objects in statechart diagrams*, page 63).

Resizing and positioning objects in statechart diagrams

To align and resize selected objects, you use the buttons on the Size toolbar, or choose Format>Size on the menu. The objects will be resized and aligned according to the object last selected (marked with black squares).

MOVING A COLLECTION OF OBJECTS

You can move a collection of objects in a statechart diagram as follows:

- 1 Select the objects to be moved (see *Selecting a collection of objects*, page 62).
- 2 When the objects are marked with squares, place the cursor on one of the objects so the cursor changes shape to a Move cursor ().
- 3 Press and hold down the left mouse button. Drag the objects to where you want to place them and release the mouse button.

Printing statechart diagrams

You can print statechart diagrams from the Designer as follows:

- 1 Open your visualSTATE Project.

- 2 On the menu, choose File>Page Setup. A dialog box will be displayed. See *Figure 49*, page 64.

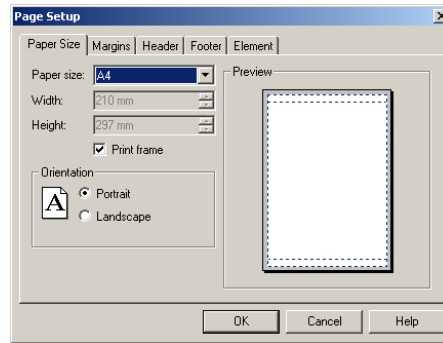



Figure 49: Designer Page Setup dialog box

Click the appropriate tabs to set up page layout.

Under the Header and Footer tabs, click the  button to insert the appropriate text.

Click **OK**.

- 3 To print a single diagram, open the diagram. On the menu choose File>Print.
- 4 To print all diagrams in the Project, choose File>Print All on the menu.

To print the full documentation for a visualSTATE Project, see *Part 8: Documenting visualSTATE Projects*, page 255.

Safe mode

If during model design you want to receive a warning when you create or use a non-verifiable element, you can use safe mode. Safe mode is set by clicking the Safe Mode button on the Standard toolbar, or choosing Tools>Safe Mode.

For information about non-verifiable elements, see *Non-verifiable elements*, page 119, and *Designing for verification*, page 143.

Customizing the Designer

The Designer can be customized with regard to handling of files, elements, message display, etc., as follows:

- I On the Designer menu, choose Tools>Settings. A Settings dialog box is displayed.

- 2 Click the appropriate tabs to set options.

The settings are stored in the registry.

CHANGING GRAPHICAL SETTINGS

You can change the graphical settings in the Designer as follows:

- 1 On the Designer menu, choose Tools>Customize. A dialog box is displayed. In the tree, click the node of a category to expand it. Select the item to be customized. See example in *Figure 50*, page 65.

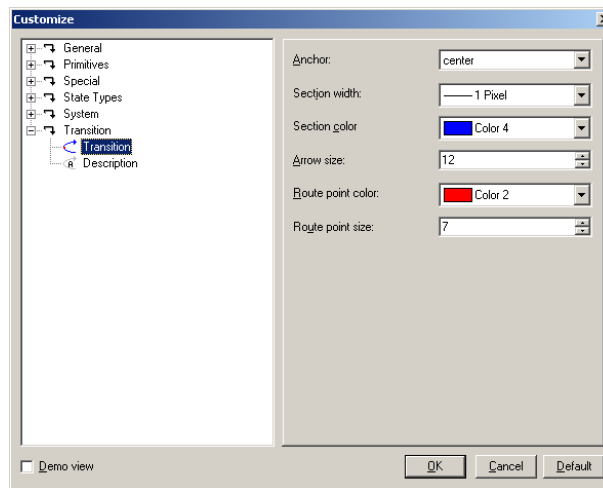


Figure 50: Designer Customize dialog box, transition category selected

- 2 In the right pane, select the values to apply.

Click the **Default** button to restore the original settings.

The settings are stored in the registry.

- 3 You can change the appearance of an individual object by selecting it in the statechart diagram and using the property window (see *Property window*, page 52).

States

This chapter describes how to compose and edit states using the Designer, including:

- Composite states
- Regions
- Connector states
- Pseudostates.

The types of states are described in detail in *visualSTATE Reference Guide*.

For a description of how to create and define state reactions, see *Elements*, page 89.

Composing states

When you have drawn a state, you can compose it with state reactions. You are recommended to use the element browser for creating elements (see *Creating and editing elements*, page 89).

To compose a state:

- I Launch the Designer, and open your visualSTATE Project.

- 2 In the statechart diagram view, double-click the state to compose. Or click the state, open the pop-up menu, and choose **Compose State...**. A Compose State dialog box is displayed. See *Figure 51*, page 68.

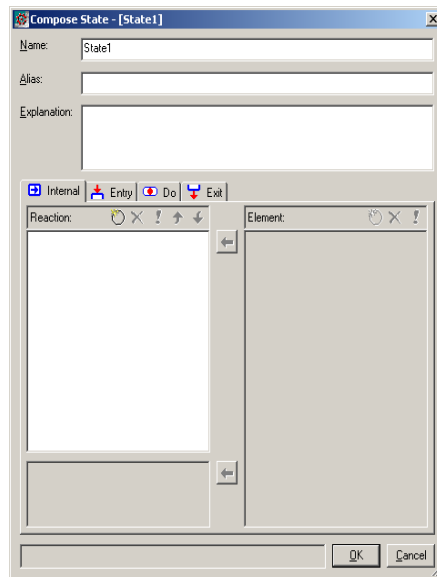



Figure 51: Compose State dialog box

Here you can change state name, and enter alias name and explanation for the state.

- 3 Go to the Reaction section. The section has tabs that represent the various categories of state reactions: Internal reaction, entry reaction, exit reaction, and do reaction.

The toolbar is used for creating, deleting, defining, and moving state reactions up or down in the list.

- 4 Click the tab containing the type of state reaction you want to use, for example internal reaction.
- 5 Then click the New button () on the toolbar. A list of state reactions is displayed in the Reaction section.
- 6 In the Reaction section, click the element type to use. For example click Trigger in the list. If state reactions have been defined, a list of the defined triggers is displayed in the Element section. See *Figure 52*, page 69 where three events have been defined.

- 7 In the Element section, double-click the element to use. The element will be moved to the Reaction section and applied to the state reaction. See *Figure 52*, page 69.

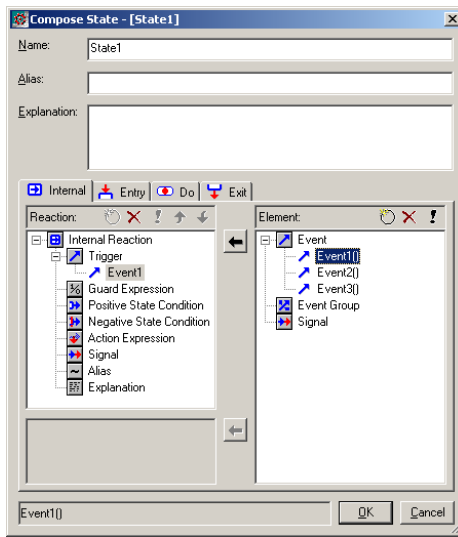


Figure 52: Compose State dialog box, Event1 added

You can add as many reactions as you want to, and change their order in the Reaction list by clicking the Up Arrow and Down Arrow buttons on the Reaction toolbar.

To edit elements in state reactions, use the element browser as described in *Creating and editing elements*, page 89.

To delete a state reaction from a state, select the reaction to delete in the Reaction section of the Compose State dialog box, and click the Delete button on the toolbar (see *Figure 52*, page 69).

For information on how to add assignments and guard expressions to state reactions, see *Adding assignments and guard expressions*, page 94.

CREATING ELEMENTS WHILE COMPOSING STATES

If no elements have been created in the element browser, you can use the Compose State dialog box, as follows:

- 1 In the statechart diagram, select the state, and open the Compose State dialog box.
- 2 Go to the Reaction section. and click the tab containing the type of state reaction you want to add, for example Internal Reaction.

- 3 Then click the New button on the toolbar to insert a new state reaction. See *Figure 53*, page 70.

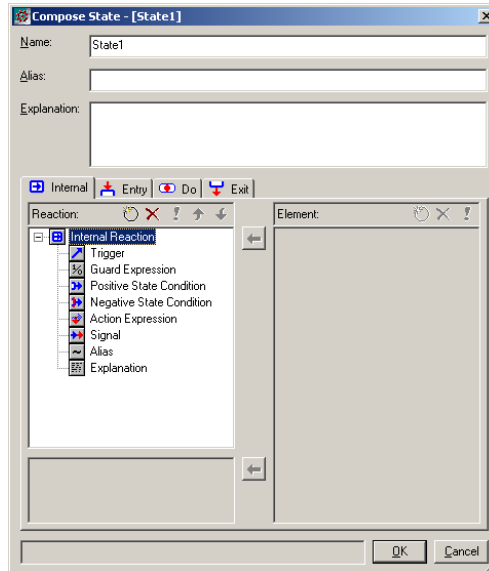


Figure 53: List of elements (Designer)

- 4 Click an element type to add. For example click Trigger in the list. A list of available triggers will be displayed in the Element section to the right.

- 5 In the Element section, click element type to create, for example Event, and click the New button in the Element section. A New Event dialog box is displayed. See *Figure 54*, page 71.

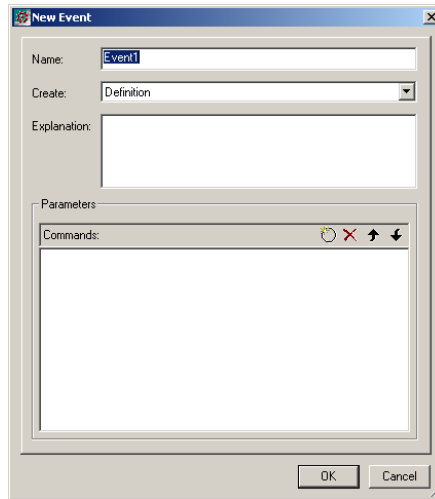


Figure 54: New Event dialog box (Designer)

Here you can type a new name for the event, enter an explanation, and specify whether the event is a definition or declaration. In the Parameters section, you can specify parameters. See *Creating parameters*, page 90.

- 6 When you have defined the event, click **OK**. The event is added to the list of elements.

- 7 Double-click the event in the Element section to move it to the Reaction section and apply it to the state reaction. See *Figure 55*, page 72.

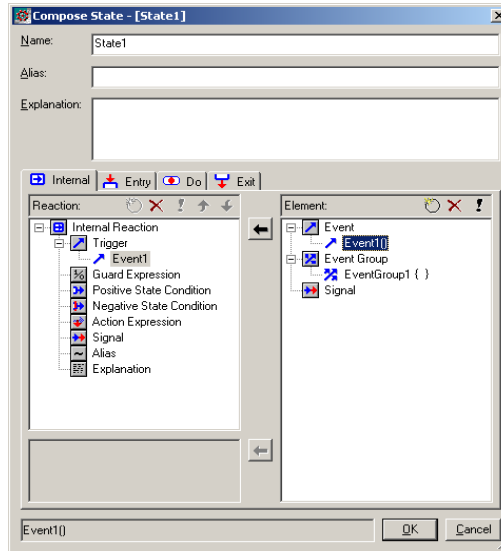



Figure 55: Compose State dialog box, event created and used as trigger

Composite states

Composite states consist of concurrent regions, or mutually exclusive substates.

CREATING A COMPOSITE STATE CONSISTING OF CONCURRENT REGIONS

To create a composite state consisting of concurrent regions:

- 1 Launch the Designer, and open your visualSTATE Project.
- 2 On the Designer Diagram toolbar, click the Composite State button ()

- 3 Go to the statechart diagram and click on it. A state with one region is created. See *Figure 56*, page 73.

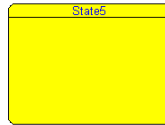


Figure 56: Composite state with one region

Note: In a state with only one region, the region has no name.

- 4 Deactivate the Composite State tool by right-clicking the mouse.
- 5 To add a region to the state, right-click anywhere in the region to open the pop-up menu. Select **Insert Region**, and select **Above**, **Below**, etc., whichever is appropriate (see *Figure 62*, page 76). The region will be inserted. See example in *Figure 57*, page 73.

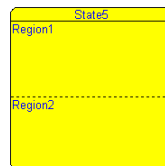


Figure 57: Composite state with two concurrent regions

The composite state can be resized and moved as necessary. You can change the sizes of the individual regions by dragging the dashed separator line between the regions.

- 6 To compose the state, click in the upper area of it (not in one of the regions), open the pop-up menu and choose **Compose State...** See *Composing states*, page 67.

Inserting already created states in a concurrent region

- I In the diagram, not on the states to be moved, click the left mouse button, hold it down and drag a box (dotted line) around the states.

- 2 Release the mouse button. The states will be marked with squares. See *Figure 58*, page 74.

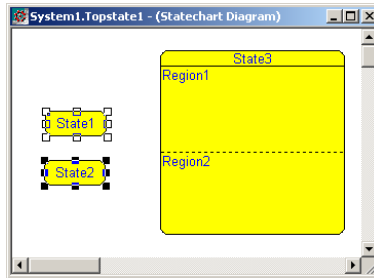


Figure 58: Selection of states to be moved (Designer)

- 3 Place the cursor on one of the selected objects. A Move cursor appears. Press and hold down the left mouse button, and drag the states into the region.
- 4 Release the mouse button to place the states in the region.

CREATING A COMPOSITE STATE CONSISTING OF MUTUALLY EXCLUSIVE SUBSTATES

To create this type of composite state, do the following:

- 1 In the diagram, select the states that are to be substates, and release the mouse button (see *Selecting a collection of objects*, page 62).
- 2 Drag the states onto the state to be the composite state. A region is automatically inserted, indicated by a horizontal line below the state name. An example of a composite state is shown in *Figure 59*, page 74.

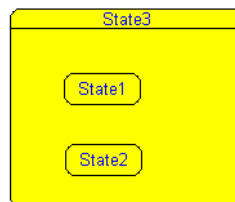


Figure 59: Composite state consisting of mutually exclusive substates

You compose the composite state as described in *Composing states*, page 67.

A composite state composed of mutually exclusive substates can be changed to a composite state with concurrent regions just by adding a region (see *Creating a*

composite state consisting of concurrent regions, page 72). This will automatically create two regions and move the original substates into one of the regions.

Regions

Regions are used in states and topstates to define concurrent subsystems and represent hierarchical state machines. For a detailed description of topstates and regions, see *visualSTATE Reference Guide*.

VIEWING CONTENTS OF REGIONS

You can choose whether you want to view the contents of a region or not. Hiding the contents of a region can give you a better overview of the overall structure of your model if the region contains a very complex model.

You hide the statechart contained in a region by opening its pop-up menu and selecting **Off-page**. *Figure 60*, page 75 shows an example of a state region containing a state machine model.

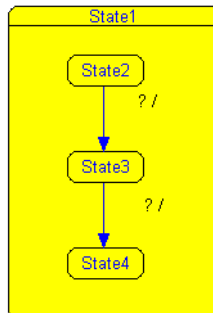


Figure 60: Example of state with one region

When the same region is off-page, it looks as follows:

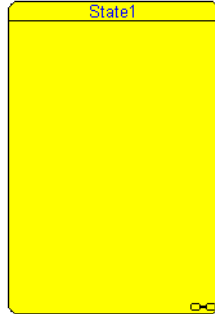


Figure 61: Off-page state region

To go to the statechart contained in a state region or topstate region, you click the statechart icon in the lower right corner (see *Figure 61*, page 76). To return from the contained statechart to the region, press the BACKSPACE key.

REGIONS IN TOPSTATES

You create regions in topstates as follows:

- 1 In the Designer Project browser, double-click a topstate.
- 2 Go to the System view and click the topstate. Open the pop-up menu and choose Insert Region>.... See *Figure 62*, page 76.

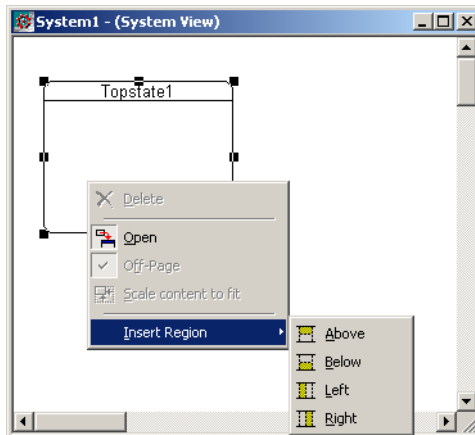


Figure 62: System view pop-up menu (Designer)

Connector states

Connector states are pairs of graphical symbols for splitting a transition into multiple transition fragments. The transition can originate from and enter a connector state.

To draw a connector state:

- 1 Open your visualSTATE Project in the Designer.
- 2 On the Diagram toolbar, click the Connector State button, go to the statechart diagram, and click where you want to insert the connector states.
- 3 Draw a transition from the connector states to ordinary states. An example is shown in *Figure 63*, page 77.



Figure 63: Example of a pair of connector states

Note: You must rename the states in a connector pair to the same name in order to have them connected. You rename a connector state by clicking it and typing a new name; press ENTER to finish.

To find the other connector state in a pair, click on the connector state, open the pop-up menu, and choose *Go to Buddy*.

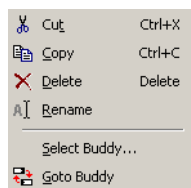


Figure 64: Connector state pop-up menu

To connect the selected state connector with another, choose *Select Buddy* from the pop-up menu.

Pseudostates

This section describes how to create pseudostates in a statechart diagram. The following pseudo states can be created:

- Initial, shallow history, and deep history states.

- Fork and join states
- Junction states.

INITIAL, SHALLOW HISTORY, AND DEEP HISTORY STATES

You draw initial, shallow history, and deep history states as follows:

- 1 Open your visualSTATE Project in the Designer.
- 2 On the Diagram toolbar, click the Initial State, Shallow History State, or Deep History State button, go to the statechart diagram, and click where you want to insert the pseudo state.
- 3 Draw a transition from the inserted pseudo state to the state that is to be the default state, in the same way as you draw transitions between states. See *Figure 66*, page 79.

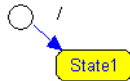


Figure 65: Example of a state with an initial state

FORK AND JOIN STATES

Fork and join states are used to go to/from multiple state machines to/from a single state machine. Fork and join states can be used across several state levels.

To draw fork and join states:

- 1 Open your visualSTATE Project in the Designer.
- 2 On the Diagram toolbar, click the Fork or Join button, go to the statechart diagram, and click where you want to insert the pseudo state.

- 3 Now you can draw transitions from states to the fork state, and from the join state to states. See example in *Figure 66*, page 79.

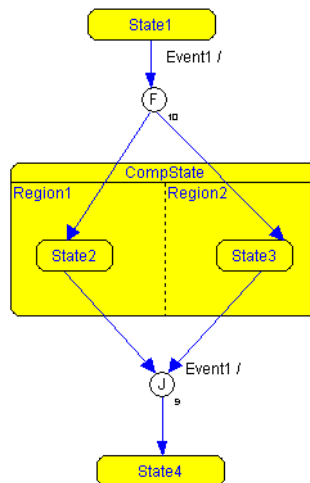


Figure 66: Example of fork and join states

The numbers automatically inserted on the bottom-right side of the pseudo states serve as indexes that are used by the visualSTATE documentation and verification tools (see *Part 4: Formal testing*, page 107, and *Part 8: Documenting visualSTATE Projects*, page 255).

You can choose to hide these numbers in the Designer diagram as follows:

- 1 Choose Tools>Settings.
- 2 In the dialog box displayed, click the Pseudo State tab. Deselect **Show index**.

JUNCTION STATES

Junction states are used to chain together and split transitions.

To draw a junction state:

- 1 Open your visualSTATE Project in the Designer.
- 2 On the Diagram toolbar, click the Junction State button, go to the statechart diagram, and click where you want to insert the junction state.
- 3 Draw transitions to and from the junction state from and to the other states in the statechart.

Excluding states and regions

States and regions can be excluded from further processing. Any number of states or regions, on any hierarchy level, can be marked for exclusion.

MARKING STATES OR REGIONS FOR EXCLUSION

To mark a state or region for exclusion, right-click in it and choose **Exclude** from the context menu:

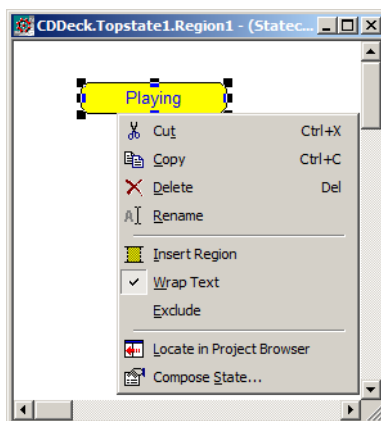


Figure 67: StatePopup

Excluded states and regions have (excluded) after their name in the diagram and in the Project Browser window.

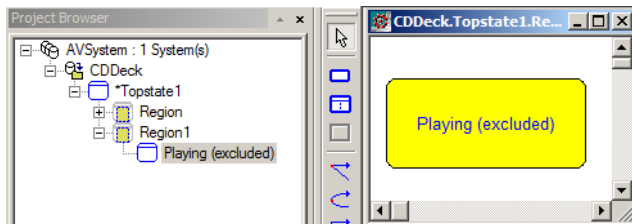


Figure 68: StateExclusion

Exclusion is inherited; all states or regions that are contained inside an excluded state or region are also excluded. (Note that an *explicitly* excluded state below a state/region that is marked for exclusion will still be excluded even if the state above is once again included.)

TRANSITIONS

A transition is ignored if it is has:

- A source state or region that is excluded
- A target state or region that is excluded
- A positive state condition that depends on an excluded state
- Both a main target and a main source that is below the top-level exclusion.

Transitions that have a negative state condition that depends on an excluded state will simply have that negative state condition removed. All other transitions are handled as if the state or region is not part of the model.

OVERRIDING EXCLUSION MARKS

At the time of code generation, validation, or verification, you can choose to include states and regions that are marked for exclusion, despite the exclusion mark. This is useful for:

- Configuring your application.

For example, you can include or exclude parts of your design to enable or disable a certain function in your application.

- Adding debug regions to your design to keep track of or detect, for example, error conditions.

Just add a separate region where you put your debug state machines and then decide if you want the functionality included in simulation, in the generated code, or for verification. The verification, in particular, can greatly benefit from this way of working, letting you, for example, create regions that should enter a dead-end state on certain conditions. (For an explanation of verification concepts, see *Part 4: Formal testing*.)

Excluding states and regions

Transitions

This chapter describes how to compose and edit transitions using the Designer.

For a description of how to create and define transition elements, see *Elements*, page 89.

For a detailed description of the visualSTATE transition elements, refer to *IAR visualSTATE Reference Guide*.

Composing transitions

A transition is composed of a condition side and an action side. When all conditions are fulfilled, the transition will be triggered, and all actions defined on the action side will be executed. Action side and condition side of transition are described in *visualSTATE Reference Guide*.

When you have drawn a transition, you can compose it with elements. It is recommended that you use the Designer element browser for creating transition elements (see *Creating and editing elements*, page 89).

To compose a transition:

- I Launch the Designer, and open your visualSTATE Project.

- Click the transition, open the pop-up menu, and choose *Compose Transition*. A Compose Transition dialog box will be shown. See *Figure 69*, page 84.

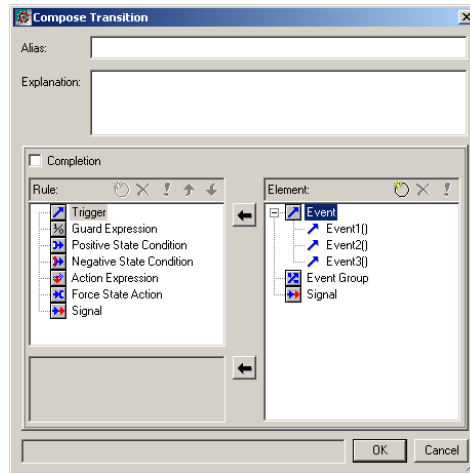


Figure 69: Compose Transition dialog box

Here you can type an alias name and explanation for the transition.

The Element section shows a list of the element types available for composing a transition:

- Triggers, guard expressions, positive state conditions, and negative state conditions belong on the condition side of the transition.
- Action expressions, force state actions, and signals belong on the action side of the transition.

- Click an item in the Rule section to display a list of the defined elements in the Element section. In the Element section, double-click the element to add (or select it and click the Left Arrow button). The selected element will be added to the transition description in the Rule section.

You can add as many elements as you want to, and change their order in the Rule list by clicking the Up Arrow and Down Arrow buttons on the toolbar.

To delete an element from a transition, select the element to delete in the Rule section of the Compose Transition dialog box, and click the Delete button on the toolbar (see *Figure 71*, page 86).

To edit elements, use the element browser as described in *Creating and editing elements*, page 89.

For information on how to add assignments and guard expressions to a transition, see *Adding assignments and guard expressions*, page 94.

CREATING A TRANSITION ELEMENT

If no transition elements have been created in the element browser, you can use the Compose Transition dialog box, as follows:

- 1 In the statechart diagram, select the transition, and open the Compose Transition dialog box.
- 2 Go to the Rule section and select the type of element you want to add, for example Trigger.
- 3 In the Element section, click the element type to create, for example Event, and click the New button. A New Event dialog box is displayed. See *Figure 70*, page 85.

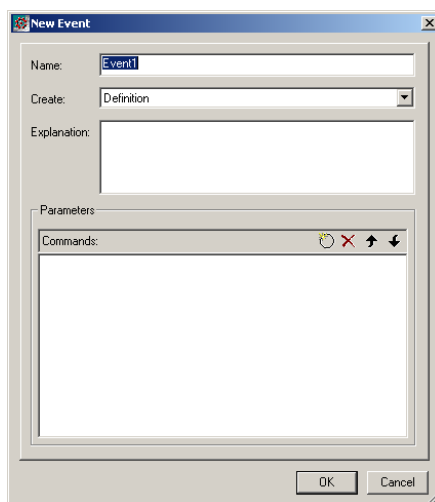


Figure 70: New Event dialog box (Designer)

Here you can type a new name for the event, and enter an explanation. Specify whether the event is a definition or declaration. In the Parameters section, you can specify parameters. See *Creating parameters*, page 90.

- 4 When you have defined the event, click **OK**. The event is added to the list of elements.

- 5 Double-click the event in the Element section to apply it to the transition. See *Figure 71*, page 86.

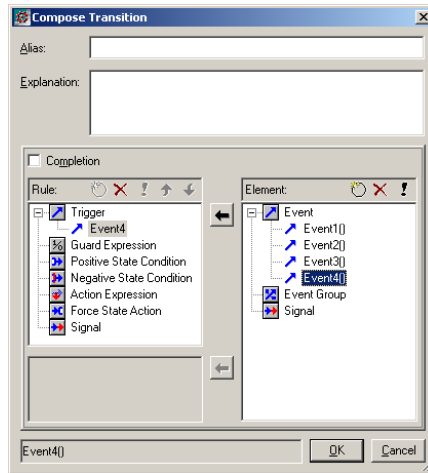


Figure 71: *Compose Transition* dialog box, event created and used as trigger

Completion transitions

A completion transition is a transition that is triggered implicitly when the last of the regions and do reactions of a composite state enters its final state.

To create a completion transition in the Designer:

- 1 Go to the statechart diagram view, and select the transition to compose.
- 2 Open the pop-up menu and choose *Compose Transition....*

- 3 In the Compose Transition dialog box displayed, select Completion. See *Figure 72*, page 87.

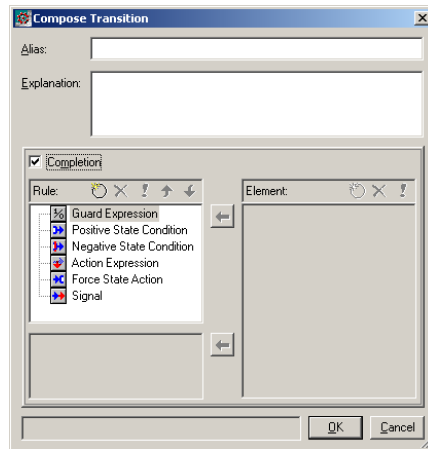


Figure 72: Completion transition selected

Elements

This chapter describes how to create, define, edit, rename, and delete elements for state reactions and transitions in the Designer. It also describes how to search for specific elements.

For a detailed description of the visualSTATE elements, see *visualSTATE Reference Guide*.

Creating and editing elements


Elements are handled via the Designer element browser where you create, define, edit, rename, and delete elements. It also gives a complete overview of the elements created for the Project. You open the element browser by selecting View>Element Browser.

When you have created a collection of elements in the element browser, you can apply them to state reactions and transitions (see *Composing states*, page 67, and *Composing transitions*, page 83).

visualSTATE elements can be local or global. Global elements are events, event groups, action functions, timer action functions, external variables, and constants. Local elements are events, event groups, action functions, timer action functions, signals, internal variables, external variables, and constants.

To create an element:

- 1 Launch the Designer, and open your visualSTATE Project.
- 2 Open the element browser window.
- 3 Go to the Project pane and select Project in the tree if you want to create a global element. To create a local element, select the topstate in the tree where you want to create the element.
- 4 Go to the Commands section of the element browser. The tabs represent the visualSTATE elements you can create. Click a tab, for example Event.

- 5 On the Commands toolbar, click the New button () A new event with a default name is created in the list, and a corresponding edit window is displayed to the right. See example in *Figure 73*, page 90.

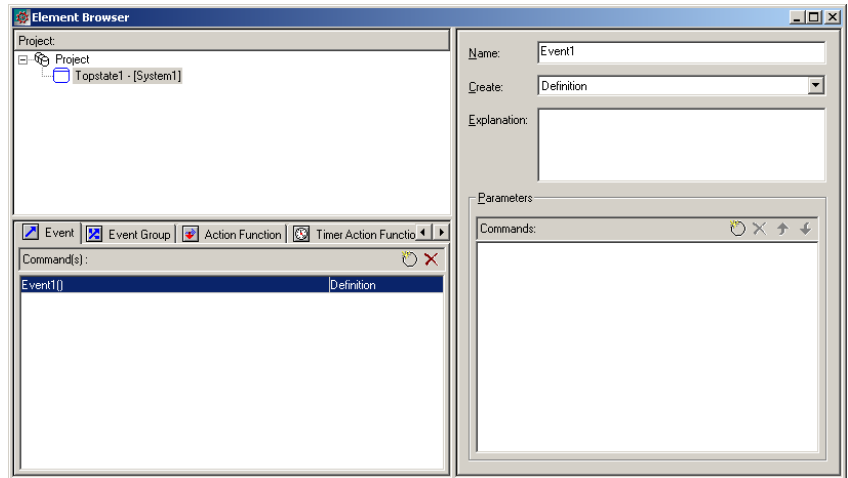



Figure 73: Designer element browser, with event created (local element)

- 6 In the edit window, you can type event name and explanation for the event, and specify whether it is a definition or declaration.

Already created elements can be dragged from the Commands section to the Project or topstates in the tree in the Project section. Thus local elements can become global elements by dragging them the Project in the tree.

You delete elements by clicking the Delete button () on the Commands toolbar.

CREATING PARAMETERS

To create a parameter for an event or action function

- 1 Open the element browser, click the Event or Action Function tab, and select the event or action function (see *Figure 73*, page 90).
- 2 Go to the edit window and click the New button in the Parameters section. A parameter is created. Click the parameter list box to select parameter type.

To delete parameters for events and action functions, click the Delete button on the Commands toolbar.

ACTION FUNCTIONS

Action functions are created and defined with return types and parameters in the Designer element browser window (see *Creating and editing elements*, page 89).

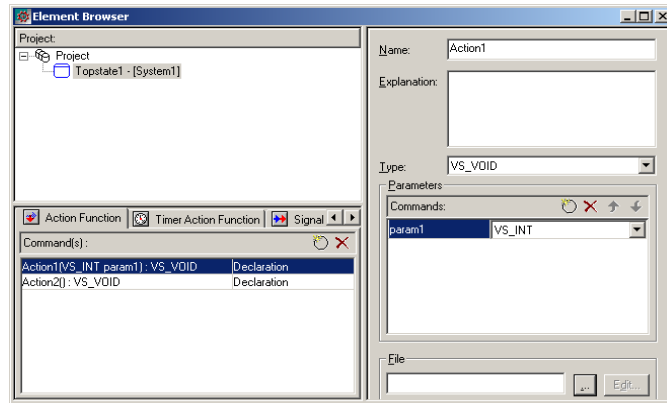


Figure 74: Defining action function

To specify return types, click the Type list box, and select the type to use (see *Figure 74*, page 91).

To create a parameter for the action function, click the New button in the Parameters section. A parameter is created. Click the parameter list box to select parameter type.

Specifying arguments for action function parameters

When an action function has been defined with parameters, you can specify the actual arguments in the individual transitions and/or state reactions where the action function is used, as follows.

- 1 Double-click the state or description of the transition to edit. The Compose State or Compose Transition dialog box respectively is displayed. Here the use of the latter will be shown, but the same procedure applies to the Compose State dialog box.

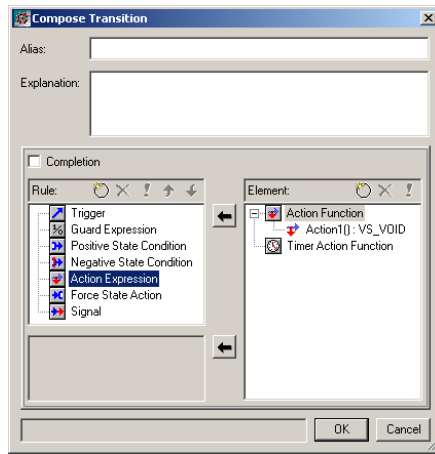


Figure 75: Compose Transition dialog box, action function

- 2 In the Rule section, select Action Expression to display the elements created in the right section. See Figure 75, page 92.
- 3 In the Element section, select the element to add, for example action function, and double-click it (or click the Left Arrow button). The element will be added to the Rule section.
- 4 Double-click the action function just added. A Define Action Function Parameter(s) dialog box is displayed. See Figure 76, page 92.

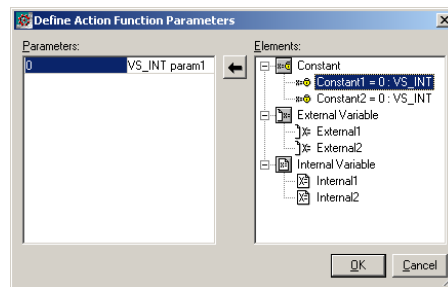


Figure 76: Define Action Function Parameters dialog box

- 5 You can now specify the parameter. Double-click Constant, External Variable or Internal Variable in the Element section to expand the tree. Select the item to use as argument for the parameter and double-click it.

Declaring action functions in external C files

Action functions are declared and implemented in an external C file, as follows:

- 1 Open the element browser, click the Action Function tab, and select the action function for which a C file declaration is to be used.
- 2 Go to the bottom of the edit window and click the Browse button. An Open file dialog box is displayed. Specify the C file to use. The C file name is displayed. See *Figure 77*, page 93.

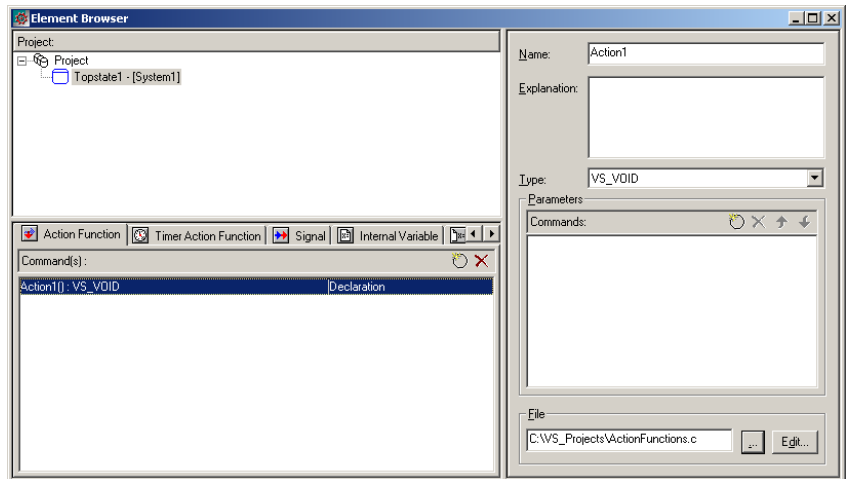


Figure 77: External C file specified for action function

- 3 Click the **Edit** button to open the C file for editing. You can set up default editor to use. See *Setting up external editor for action functions*, page 93.

Setting up external editor for action functions

It is possible to edit action functions with an external editor from within the Designer. Default editor is *IAR Systems IAR Edit*.

You can set up editor as follows:


- 1 On the menu, choose Tools>Settings. A Settings dialog box is displayed.

- 2 Click the External Editor tab.
- 3 Specify external source code editor.

ADDING ASSIGNMENTS AND GUARD EXPRESSIONS

Assignments and guard expressions are added to state reactions and transitions using the Compose State and Compose Transition dialog boxes. This section describes how to add guard expressions and assignments to a state reaction, but the same procedure applies to transitions.

To add an assignment or guard expression to a state reaction:

- 1 In the statechart diagram, select the state and open the Compose State dialog box.
- 2 Click the appropriate tab and select Guard Expression or Action Expression, whichever you want to apply. The defined elements are listed. If you need a new assignment or guard expression click the New button () on the Commands toolbar.
- 3 Go to the Element section and double-click the item to apply, for example internal variable for a guard expression. The item will be moved to a section below the Reaction section. See *Figure 78*, page 94.

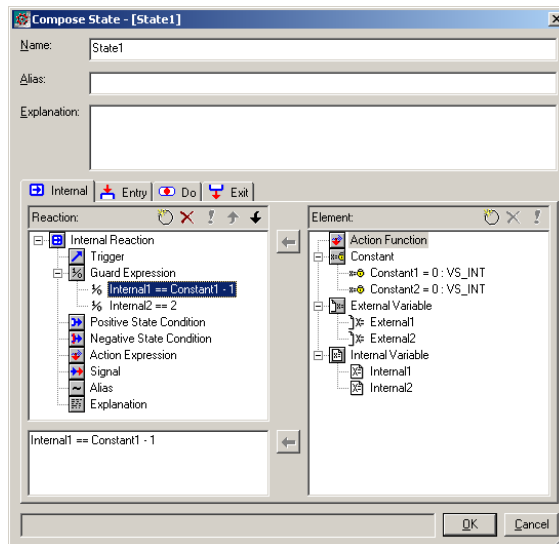


Figure 78: Compose Transition dialog box, guard expression value

- 4 Go to the edit field of the Reaction section, and enter a value for the assignment or guard expression selected, according to the syntax for assignments and guard expressions (described in *IAR visualSTATE Reference Guide*).
- 5 Press the ENTER key, or click another item to accept the assignment or guard expression changes.

SPECIFYING SIGNAL QUEUE BEHAVIOR

Because the type of signal queue influences verification and code generation, it is possible to specify signal queue behavior, as follows.

- 1 Launch the Designer, and open your visualSTATE Project.
- 2 In the Project browser, click the System view tab, and select Project. Open the pop-up menu and choose *Compose*.
- 3 In the Compose Project dialog box displayed you can specify signal queue behavior, either *Drop if full*, or *Error if full*.

SPECIFYING SIGNAL QUEUE SIZE

You specify signal queue size as follows:

- 1 Launch the Designer, and open your visualSTATE Project.
- 2 In the Project browser, select the appropriate System. Open the pop-up menu and choose *Compose...*
- 3 In the Compose System dialog box displayed, specify signal queue length. See *Figure 82*, page 102.

Searching for an element

It is possible to search for a specific element to find out in which transitions and state reactions of the model it is used, as follows:

- 1 Launch the Designer, and open your visualSTATE Project.
- 2 Click the Find button on the Standard toolbar, or choose Edit>Find... on the menu. A Find dialog box is displayed.
- 3 In the Find what field, type the name of the element to find, and select the appropriate Include options.
- 4 Click *Find* to start the search.

The result of the search is shown in the output window (Find tab). An icon shows where the element was found, and a description is given.

Handling Projects, Systems, and files for modeling

This chapter describes how to handle Projects, Systems and files in the Designer. It also describes the following:

- Specifying number of System instances.
- Using Designer backup files.
- Using function declarations and constants in existing files.

Creating and saving Projects, Systems, and files in the Designer

You can create visualSTATE Projects in the Designer with statechart files and Systems. The Project created in the Designer can later be imported into a Navigator workspace. For information about import of Projects into the Navigator, see *Adding an existing Project to a workspace*, page 27.

Files created in the Designer can be added to a source code control system using the File>Source Control commands on the main menu, or the Source Control toolbar (see *Source Control toolbar*, page 54). For a description of using source code control, see *Source code control*, page 37.

To create and save a Project with Systems and Statechart files:

- 1 Choose File>New on the menu, or click the New button. See *Figure 79*, page 98.

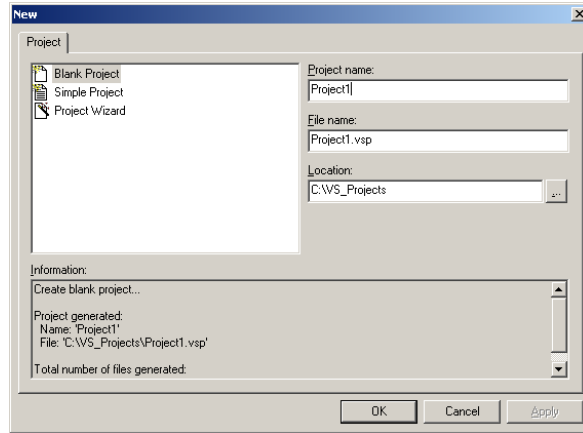


Figure 79: New dialog box

- 2 Select one of the following:

To create a Project without Systems, select **Blank Project**. Go to *Creating Systems and Statechart files in a blank Project*, page 99.

To create a simple Project with one System and one topstate, select **Simple Project**. Specify Project name, Project file name (extension `vsp`), and location of Project file.

To create a customized Project, select **Project Wizard**. The use of this wizard is described in *Creating a Project using Project wizard*, page 27).

- 3 When you have created the Project, choose File>Save Project.

CREATING SYSTEMS AND STATECHART FILES IN A BLANK PROJECT

When you have selected *Blank Project* in the New dialog box (see *Figure 79*, page 98), the Project will be created in the Designer, as shown in *Figure 80*, page 99.

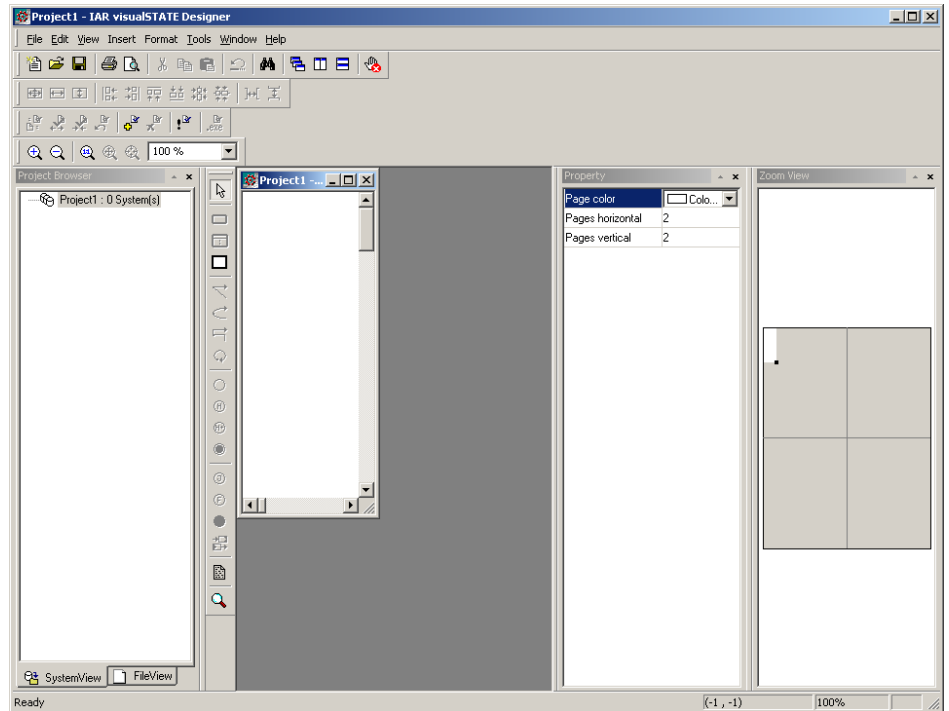




Figure 80: Designer with blank Project

In order to be able to create statechart diagrams in the Project, you must create a System and Statechart file, as follows:

- 1 To create a System, click the System button on the Diagram toolbar (). The tool becomes active.
- 2 Move the cursor to the Project view, and click in the window. A square will be inserted representing the new System, and the Project browser will be updated with the System. Right-click to deactivate the tool.
- 3 In the System drawn, not on the System name, double-click. The System view is opened.

- 4 Click the Composite State button () on the Diagram toolbar. The tool becomes active.
- 5 Move the cursor to the System view, and click in the view. A topstate will be inserted representing a new statechart file which will contain the statechart diagram. The Project browser will be updated. Right-click to deactivate the tool.
- 6 In the topstate drawn, not on the topstate name, double-click. The statechart diagram view is opened. See *Figure 81*, page 100

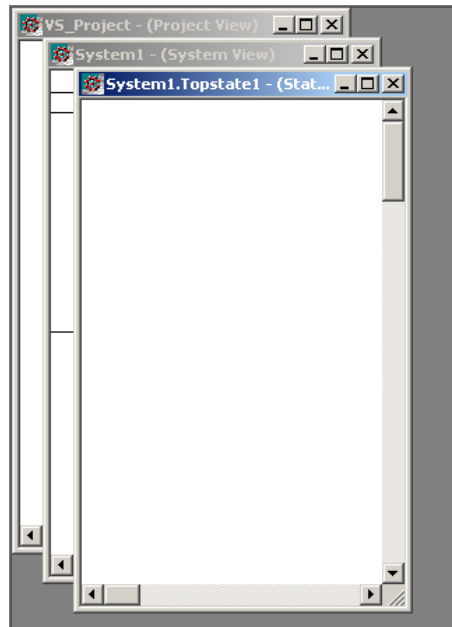


Figure 81: Diagram window, with empty statechart diagram

You can now start designing the statechart diagram with states and transitions. See *Designing statechart diagrams*, page 57.

- 7 On the Designer menu, choose File>Save Project.

CREATING A COPY OF A STATECHART FILE

You can create a copy of a Statechart file (`vsr` file) in the Designer, as follows:

- 1 Open your visualSTATE Project.
- 2 Go to the Project browser, and open the file view. Select the `vsr` file to copy.

- 3 On the menu, choose Save As.... In the Save As dialog box displayed, specify name and directory of the new Statechart file.

Note: It is not possible to create a copy of a Project file using the Save As command.

The Statechart file can be imported to a Project as described in *Importing files into the Designer*, page 101.

Opening a Project in the Designer

You open a visualSTATE Project in the Designer by clicking the Open button on the toolbar, or choosing File>Open.

One Project at a time can be open in the Designer.

Importing files into the Designer

You can import C header files into a visualSTATE Project or topstate as follows:

- 1 Launch the Designer and open your visualSTATE Project.
- 2 Go to the File view of the Project browser and select the Project or topstate to import to.
- 3 Open the pop-up menu, and choose *Import...*
- 4 In the Import dialog box displayed, specify the file to import.

Specifying number of System instances

It is possible to create multiple instances of the same visualSTATE System. Instances are useful in the case of a System containing more than one identical state/event section, for example as when controlling ten identical devices.

To specify number of System instances:

- 1 Launch the Designer, and open your visualSTATE Project.

- 2 In the Project browser, select the System for which to specify instances. Open the pop-up menu, and choose **Compose**. A Compose System dialog box is displayed.

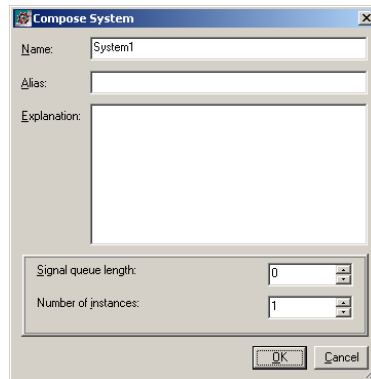


Figure 82: Compose System dialog box

- 3 Specify number of System instances.

Using Designer backup files

By default, the Designer will create backup files of the `vsp` and `vsr` files on every save of the visualSTATE Project in the Designer. You may also choose to have backup files created at regular intervals (see *Setting backup options*, page 103).

The `vsr` and `vsp` backup files are created in the directory where the visualSTATE Project is located. The backup files have the extensions `vsr.bk#` and `vsp.bk#` respectively where # is a number from 1 to 9.

It is possible to have up to nine backup files of each type created, depending on the number specified (see *Setting backup options*, page 103).

When a new backup file is created, it is given the extension `bk1`. The previous `bk1` backup file is automatically renamed to `bk2`, the `bk2` file is renamed to `bk3`, etc. Thus the latest backup file created always has the extension `bk1`.

To use a backup file for a visualSTATE Project:

- 1 Open a file browser or file management tool.
- 2 Find the `vsp.bk#` and `vsr.bk#` files you want to use, for example `AvSystem.vsp.bk1` and `CDPlayer.vsr.bk1`.

Note: The `vsr` and `vsp` files must have the same `bk` extension number, for example `vsp.bk1` and `vsr.bk1`.

- 3 In the file browser, remove the `bk#` extensions of the `vsp` and `vsr` files. In this example the files would be renamed to `AvSystem.vsp` and `CDPlayer.vsr`.
- 4 Go to the Designer, and open the `vsp` file you have renamed.

The visualSTATE Project will be loaded with the backed up `vsp` and `vsr` files.

SETTING BACKUP OPTIONS

By default, the Designer will create backup files of the `vsp` and `vsr` files on every save of the Project. If you want backup files to be created at regular intervals, you can choose to set up interval backup, as follows:

- 1 On the menu, choose `Tools>Settings`. A *dialog box will be displayed*. See *Figure 83*, page 103.

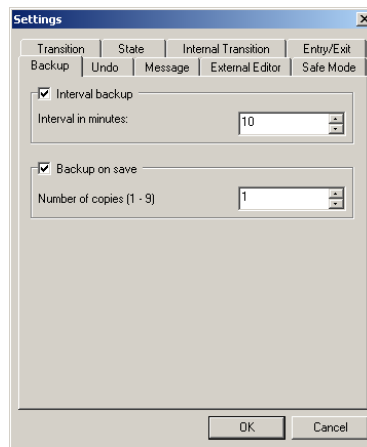


Figure 83: Settings dialog box, file backup options (Designer)

- 2 Click the Backup tab.
- 3 Select the Interval Backup check box.
- 4 Specify interval in minutes.
- 5 To specify number of backup copies on save, select the Backup on Save check box. It is possible to have up to nine copies of backup files created on save.

Using function declarations and constants in existing files

You can reuse an existing C header file containing function declarations and constants for your visualSTATE Project. This is done by importing the C header file into the Project loaded in the Designer.

Note: It is only constants contained in macros that can be imported.

The function declarations and constants must have a special syntax in order for the Designer to recognize them. This is described in *Syntax of C header files*, page 105.

The function declarations in the imported header file map to action functions in the visualSTATE model. The constants in the header file map to constants in the visualSTATE model.

Note: With IAR MakeApp, special visualSTATE MakeApp files with correct syntax for the C header files can be generated, thereby providing mapping from the IAR MakeApp device drivers to the visualSTATE logic. For detailed information about the generation of such files, refer to the IAR MakeApp user documentation.

To import a C header file:

- 1 Launch the Designer, and open your visualSTATE Project.
- 2 In the Project browser, select the Project or topstate into which to import the C header file. Open the pop-up menu, and choose **Import**. An Import dialog box will be displayed. Select the C header file(s) to import, and click **OK**.

- 3 The header file will be loaded and analyzed for function declarations and constants, and an Import Elements dialog box is displayed. See *Figure 84*, page 105.

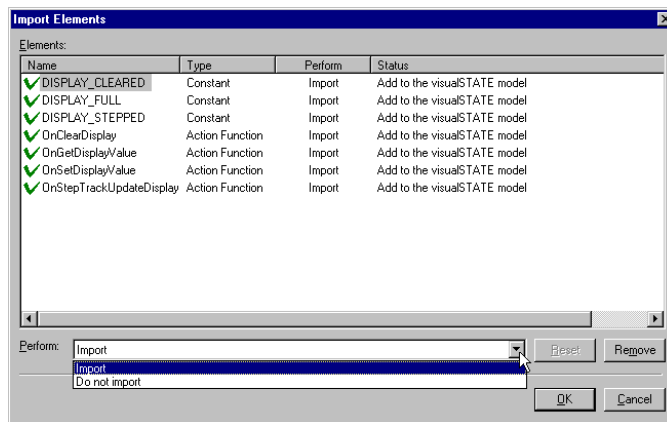


Figure 84: Import Elements dialog box (Designer)

- 4 Select the items to import and click **OK**. The selected items will be imported and displayed in the element browser.

Note: If the external C header file contains constants and action functions have the same names as those already defined for the Project or topstate, the items in question will not be imported from the external file.

SYNTAX OF C HEADER FILES

A special syntax is required for C header files containing function declarations and constants that are to be imported into a visualSTATE Project.

Syntax for import of function declarations

Import of function declarations (map to action functions in visualSTATE) can be done either by single import statement or multiple import statement:

Single import statement

```
#pragma VS_ACTION <function declaration>
```

where <function declaration> is a standard ISO/ANSI C function declaration.

Multiple import statement

```
#pragma VS_ACTION_BEGIN
  <function declaration 1>
  ...
  <function declaration N>
#pragma VS_END
```

where <function declaration 1 ... N> is a standard ISO/ANSI C function declaration.

See example in *Example of import syntax*, page 106.

Syntax for import of constants

Import of constants (map to constants in visualSTATE) is done by multiple import statement as follows:

```
#pragma VS_CONSTANT_BEGIN
  <macro statement 1>
  ...
  <macro statement N>
#pragma VS_END
```

where <macro statement 1 ... N> is a standard ISO/ANSI C macro statement.

See example *Example of import syntax*, page 106.

Example of import syntax

```
// functions to import
#pragma VS_ACTION void OnClearDisplay(void);
#pragma VS_ACTION_BEGIN
  int OnGetDisplayValue(void);
  void OnSetDisplayValue(int nValue);
  int OnStepTrackUpdateDisplay(int nStep, int nValue);
#pragma VS_END

// constants to import
#pragma VS_CONSTANT_BEGIN
  #define DISPLAY_FULL 0x01
  #define DISPLAY_STEPPED 0x02
#pragma VS_END
```

Closing the Designer

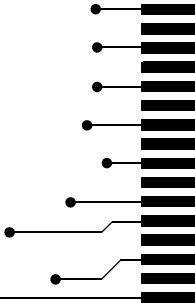
To close the Designer application, choose File>Exit.

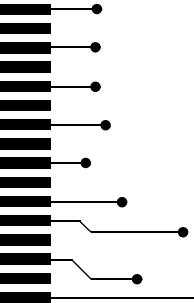
Note: All Designer instances launched from the Navigator will be closed when the Navigator is closed.

Part 4: Formal testing

This part of the visualSTATE[®] User Guide includes the following chapters:

- Introduction
- Checks performed by visualSTATE Verificator
- Verifying your visualSTATE Project
- Designing for verification.





Introduction

This chapter explains what is understood by verification in visualSTATE, and why you are recommended to use it in your development process. It describes the most important concepts related to formal verification, and gives examples of the checks that can be performed by the visualSTATE Verifier.

Conventions used in this part

In this part, the following conventions apply:

Parallel state machines contained in Systems are separated by a dashed vertical line. See example in *Figure 85*, page 111.

The state space of a System is the set of possible System configurations.

The following conventions apply to constructs:

Construct	Notation	Example
Source states	A letter followed by a colon.	B: E2 () / -> A (where B is the source state).
Destination states	-> followed by a letter.	B: E2 () / -> A (where A is the destination state).
Transitions	Action side / Condition side. For a detailed description of transition syntax, see <i>IAR visualSTATE Reference Guide</i> .	
Internal variables	Are named <i>i</i> , <i>j</i> , <i>k</i> , unless otherwise stated.	
External variables	Are named <i>x</i> , <i>y</i> , <i>z</i> , unless otherwise stated.	
State configurations	A state configuration for a System is written as a tuple with one state from each machine.	(A, B, C) is the state configuration for a System with three states where A is a state in the first machine, B is a state in the second machine, and C is a state in the third machine.

Table 3: Conventions used for constructs

Construct	Notation	Example
Set of state configurations	A set of state configurations is written as a number of cross-products.	The state configurations (A, B, C), (D, B, C), and (E, F, G) will be written as $\{A, D\} \times \{B\} \times \{C\}$ $\{E\} \times \{F\} \times \{G\}$
System configuration	Is written as a state configuration extended with the values of variables and the signal queue if signal queue is not empty.	(B, D, $i = 1$, S1 S2) is a System configuration where the first machine is in B, the second machine is in D, the variable i has the value 1, and the signal queue contains the signals S1 and S2.

Table 3: Conventions used for constructs (Continued)

Verification with visualSTATE Verificator

Verification with visualSTATE Verificator is characterized by the following, in contrast to simulation with visualSTATE Validator:

- Formal test: the logical consistency of a visualSTATE Project is checked. The Verificator does not test functionality, in contrast to visualSTATE Validator.
- Check of complex properties such as state dead ends.
- Complete examination of models with large state spaces.

Overview

When the visualSTATE Verificator performs an analysis of a visualSTATE System, it uses formal verification. Formal verification is performed by creating a formal description of a visualSTATE System and establishing properties of the System using rules of inference. This approach proves the properties of a System in the same way as theorems are proven in mathematics.

Example

In this section, the visualSTATE System shown in *Figure 85*, page 111 is used for illustrating some of the checks performed by the Verificator. A detailed description of

the Vericator checks is given in *Checks performed by visualSTATE Vericator*, page 123.

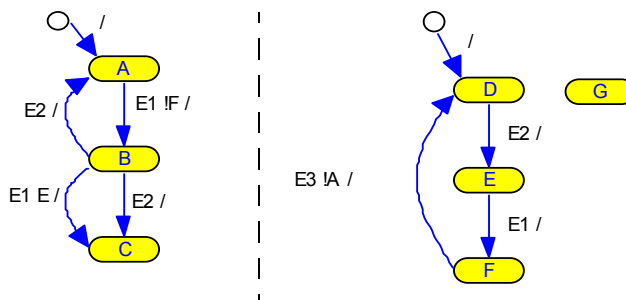


Figure 85: visualSTATE System consisting of two state machines, R0 and R1

If you run the Vericator on this System, it will report a number of results, including the following:

- Never activated elements.

The following elements will never be activated:

The state G

The transition

B:

E1 () E /

-> C

- Conflicting transitions.

Two transitions with common trigger and source state, but different destination states are said to be conflicting if they both can be triggered at the same time. The System shown in *Figure 85*, page 111, has the following conflicting transitions for event E2:

B:

E2 () /

-> C

B:

E2 () /

-> A

- State dead ends.

State dead ends are states in a state machine that once entered cannot be left. The System shown in *Figure 85*, page 111, has the following state dead end:

C

- Local dead ends.

Local dead ends are sets of states from different state machines that prevent a state machine from changing state. The System shown in *Figure 85*, page 111, has the following local dead ends:

Local dead end for the machine: R0
 $\{\text{topState.A}, \text{topState.C}\} \times \{\text{topState.F}\}$
 $\{\text{topState.C}\} \times \{\text{topState.D}, \text{topState.E}\}$

Local dead end for the machine: R1
 $\{\text{topState.A}\} \times \{\text{topState.F}\}$

- System dead ends.

System dead ends are state configurations that prevent all the state machines in the System from changing state. The System shown in *Figure 85*, page 111, has the following System dead end:

(A, F)

WARNINGS AND ERRORS

Warnings about never activated elements and dead ends might indicate errors in the model. For example because the transition in *Figure 85*, page 111

```
B:
  E1 () E /
-> C
```

is never triggered, it can be removed without changing the behavior of the model.

Whereas never activated elements and dead ends might indicate errors, conflicting transitions in a System are always an error and are reported as an error. It is not possible to generate correct code for a System containing conflicting transitions.

For a list of the warnings and error messages given by the Verificator, see *Table 4*, page 136.

Approach

The verification results described in *Overview*, page 110 are found by the Verificator after examining the complete state space of a visualSTATE System.

The Verificator derives its power from representing Systems symbolically. Instead of working on single System configurations, the Verificator works on sets of state configurations.

Treating state configurations symbolically can make verification of Systems with large state spaces possible, illustrated by the System in *Figure 86*, page 113.

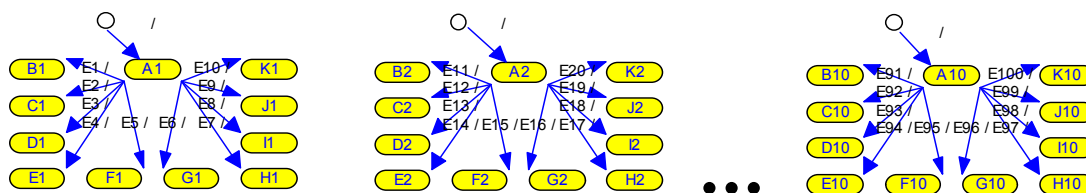


Figure 86: Example of a System with a large state space.

The System consists of ten state machines. Any of the state configurations can be reached in 10 steps making it possible to completely explore the large state space in just 10 steps.

This System has a state space consisting of 11^{10} state configurations. Using a simulation tool for checking this System is clearly not possible because of the large state space—stepping through all the state configurations would take extremely long time. Exploring the state space symbolically can be done in no more than ten steps. When the Verificator explores the state space of a System, it starts out with the initial state configuration and at each step explores all possible transitions. Therefore the entire exploration of the state space can be performed in a number steps equal to the length of the longest possible sequence of events leading to different state configurations.

Aspects of formal verification

LOGICAL CONSISTENCY

The Verificator performs a dynamic analysis of the behavior of a System to check its logical consistency.

When performing a dynamic analysis, the model is placed in an environment where any sequence of events is possible. If the model is consistent in this most extreme environment, it is also consistent in real-world environments.

In visualSTATE, logical consistency comprises the following aspects:

- Are all elements used?
- Are all elements activated?
- Are there any ambiguities?

- Does the signal queue have the right size?
- Does the System contain any dead ends?
- Does the System contain any conflicting transitions?

VERIFICATION MODES

When verifying a System, it is possible to vary the level of detail by applying one of the following verification modes:

- Full mode: Comprises verification of the entire control logic and data (see *Full mode*, page 115).
- Guard mode: Comprises verification of the entire control logic (see *Guard mode*, page 115).
- Basic mode: Comprises verification of the basic parts of the control logic (see *Basic mode*, page 118).
- Compositional mode: Basic mode verification optimized for compositional Systems (see *Compositional mode*, page 118).

Some models are too complex to be verifiable in full verification mode but can be partially checked in guard mode or basic mode.

These verification modes differ in what is considered the model, the interface, and the environment. When verifying a System, a model is created which is placed in an environment which differs according to the verification mode. See *Figure 87*, page 114. These environments exhibit any possible behavior to make sure any possible state configuration can be reached. For example, in basic mode the model uses events for interfacing to the environment which consists of any possible sequence of events.

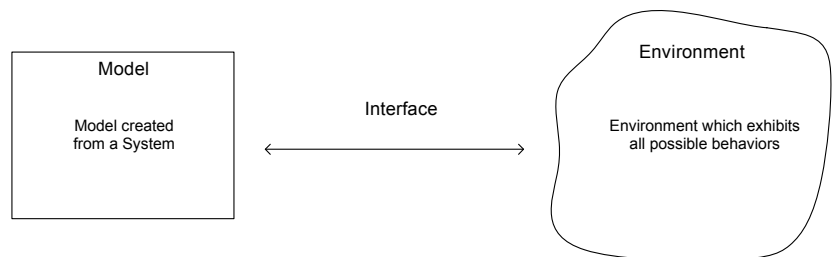


Figure 87: Model, interface, and environment.

The Verifier creates a model from a System which is then tested by placing it in an environment which exhibits all possible behaviors. What becomes the model, the interface, and the environment depends on the verification mode.

Full mode

In full mode both guard expressions and assignments are included in the verification which means that the data part of the model is also verified. Internal variables are considered part of the model, whereas events, external variables, event parameters, and action functions are considered the interface to the environment. The values of internal variables are remembered between transitions. See *Figure 88*, page 115.

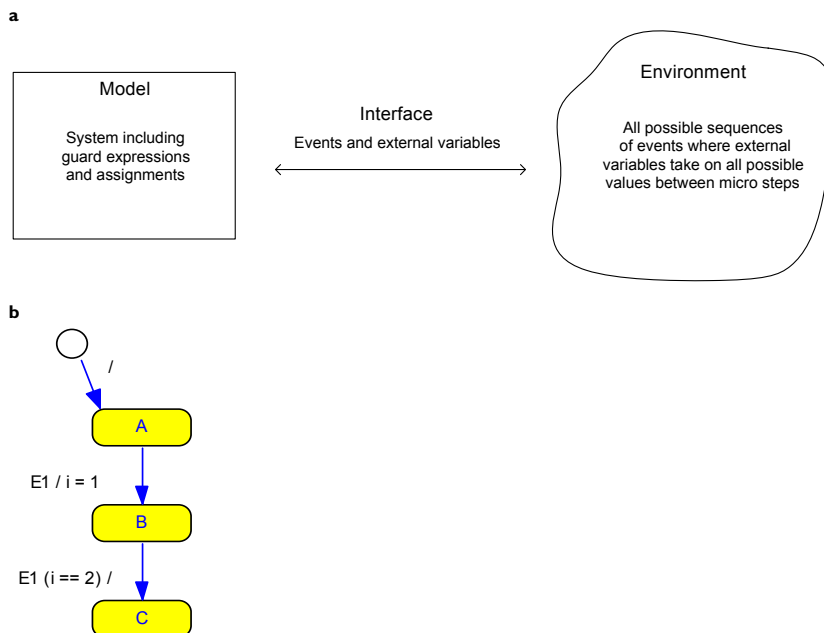


Figure 88: Full verification mode, assumptions.

In full mode, all variables, except internal, are considered part of the environment as shown in a. The values of internal variables are remembered between transitions. State C is not reachable in the System in b.

Note: If you apply full mode verification to a design with multiple assignments to the same variable, or reading and writing the same variable, an error will occur (see *Systems with ambiguous behavior*, page 119).

Note: The following element types are not verified in guard verification mode or full verification mode: VS_FLOAT, VS_DOUBLE, and VS_VOIDPTR.

Guard mode

In this mode, guard expressions are included in the verification in addition to the basic control logic, whereas assignments are not included. When evaluating guard expressions we need to make some assumptions about the values of variables (including event

parameters and action functions). The variable assumptions made for guard mode are the following:

- Between microsteps, the values of all variables can change arbitrarily.
- During each microstep, the values of variables are fixed.

Microsteps and macrosteps are described in *IAR visualSTATE Reference Guide*.

The first assumption is illustrated in the visualSTATE System shown in *Figure 89*, page 116. At A, before the first transition, the value of *i* is arbitrary. During the transition A -> B, it is assumed to be 1. At B, the value of *i* is again arbitrary. During the next transition it is 2, and at C it is again arbitrary.

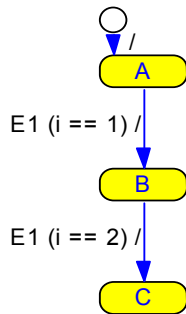


Figure 89: Guard verification mode, arbitrary values of variables between microsteps

The second assumption is illustrated in *Figure 90*, page 116. The values of variables are assumed to have some arbitrary but fixed value during each microstep. Consequently, the two transitions A -> B and D -> E can never fire during the same microstep making the state H unreachable.

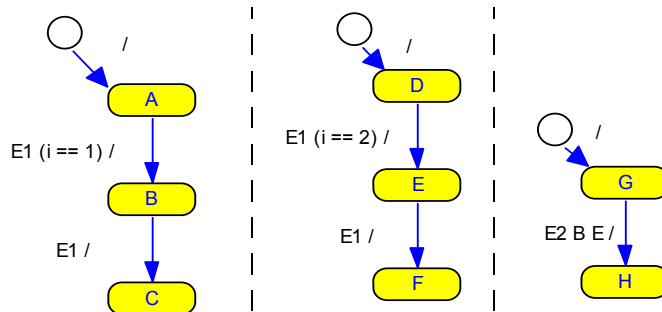
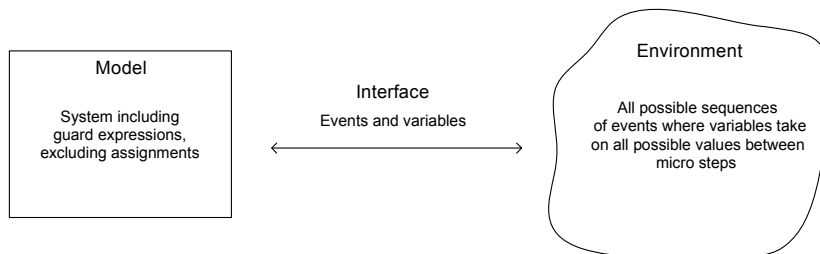


Figure 90: Guard verification mode, fixed values of variables. The System consists of three state machines. During each microstep the values of variables are fixed making the state H unreachable.

The idea is to view all variables as part of the environment. By letting the values of the variables change arbitrarily, the model is examined in any possible environment. See *Figure 91*, page 117.

a



b

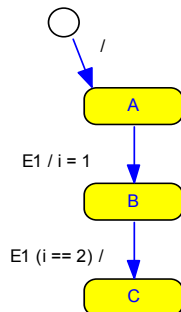


Figure 91: Guard verification mode, assumptions.

Variables are considered part of the environment in guard mode. State C in System \mathfrak{B} is reachable because the value of the variable i is not fixed between microsteps.

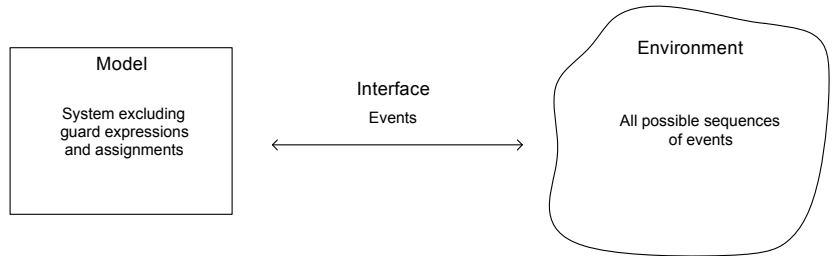
Note: The following element types are not verified in guard verification mode or full verification mode: `VS_FLOAT`, `VS_DOUBLE`, `VS_VOIDPTR`.

Note: Action functions without parameters are assumed to return the same value each time they are called during the same microstep.

Basic mode

In this mode, guard expressions are assumed to be true, and assignments are ignored. See *Figure 92*, page 118.

a



b

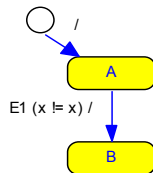


Figure 92: Basic verification mode, assumptions.

Model in an environment where any sequence of events is possible. In this mode the state B in System \mathcal{D} will not be reported as never activated because assignments and guards are ignored.

Note: This mode should only be used if your design cannot be verified in guard mode or full mode.

Compositional mode

Using this mode can speed up verification of compositional Systems, that is, Systems consisting of many independent state machines, with few state conditions and few signals.

In compositional mode, guard expressions are assumed to be true, and assignments are ignored as in basic mode. See *Figure 92*, page 118. Verification of the System starts with a possible destination state configuration and checks whether it is possible to reach the initial state configuration.

Note: This mode should only be used for compositional Systems. State dead ends and System dead ends cannot be checked in this mode (see *Table 4*, page 136).

NON-VERIFIABLE ELEMENTS

visualSTATE expressions that use arrays, `VS_FLOATS`, and `VS_DOUBLES` are non-verifiable. In the visualSTATE Designer, you can set a safe mode option by which you will be given a warning when you create or use non-verifiable elements during model design (see *Safe mode*, page 64).

SYSTEMS WITH AMBIGUOUS BEHAVIOR

The UML does not specify the sequence in which transitions are triggered, and the Verificator does not assume any specific sequence in which assignments on transitions are executed. This means that some visualSTATE Systems are ambiguous, and in such cases the Verificator will give an error message.

An exception to this rule is shown in *Figure 95*, page 121, System *c*. Both reading and writing a variable in *one* assignment is unambiguous.

The System shown in *Figure 93*, page 119 is ambiguous because the sequence in which the transitions will be triggered is not specified. Which System configuration should be entered after the event `E1`? (`B, D, i = 1`), (`B, D, i = 2`), or maybe (`B, D, i = 3`)?

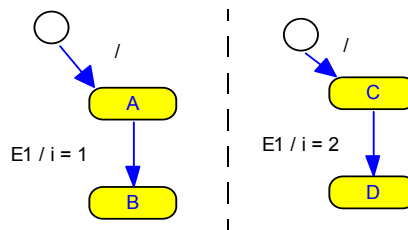


Figure 93: System with ambiguous behavior because of assignments.

The System consists of two state machines. The assignments to `i` are ambiguous.

In the System shown in *Figure 93*, page 119, the assignments to `i` are ambiguous which means that the design cannot be verified in full mode.

Another example of a design that cannot be verified in full mode is shown in *Figure 94*, page 120.

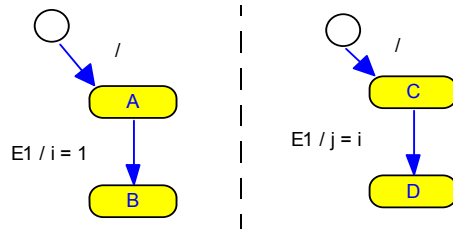
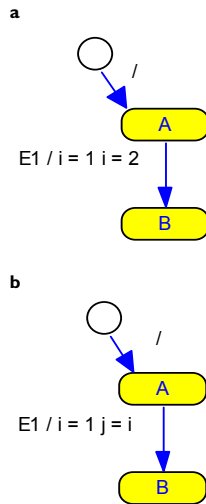


Figure 94: System with ambiguous behavior because of assignments.

The System consists of two state machines, the variable i is initialized to 0. The System is ambiguous because the sequence in which the assignments on the two transitions $A \rightarrow B$ and $C \rightarrow D$ are executed is not specified. Should the value j be equal to 1 or equal to 0 after the event $E1$?

Figure 95, page 121 shows three Systems of which only the System c has unambiguous behavior.



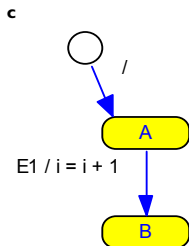


Figure 95: Three Systems of which \mathfrak{a} and \mathfrak{b} have ambiguous behavior because of assignments.

Three Systems. Multiple assignments to the same variable, or reading and writing the same variable should be avoided in full verification mode. In System \mathfrak{a} , the variable i is assigned several times. In System \mathfrak{b} , i is assigned ($i=1$) and i is read ($j=i$). System \mathfrak{c} has unambiguous behavior because i is both read and written in the same assignment.

VARIABLES, DOMAINS, AND ARITHMETICS

Non-floating-point domains in expressions and assignments can be freely mixed. Mixed domains are handled using promotion and automatic conversion the same way as in C/C++. Any cases left open as undefined or implementation-defined by the C/C++ standard are handled in the same way as by an IAR Systems compiler.

The Verifier can optionally check that type domain ranges are observed in assignments, to find cases of unintentional wrap-around. To keep the arithmetics semantics of a 16-bit target system, the size of `VS_(U) INT` can be specified as 16 bits instead of the 32-bit default. In rare cases it can be beneficial to force *all* variables and expressions in a system to have one fixed bit width. You can specify this with the option **Specify bits for encoding variables** (`-B`).

See also *Non-verifiable elements*, page 119.

CONFLICTING TRANSITIONS

The Verifier will warn about conflicting transitions.

However, depending on the verification mode you are using, various assumptions are made about the surrounding external environment. For example, as we have seen, the Basic verification mode discards assignments and assumes that every guard condition is true. This means that two transitions out of the same state, that trigger on the same event, will be reported as conflicting even if their guard conditions are mutually exclusive.

The Full verification mode might also report conflicting transitions that actually do not conflict, because it makes reasonable assumptions about, for example, function return values.

Thus you need to check conflicting transitions manually.

Figure 96, page 122 shows three Systems. In Systems a and b there are no conflicting transitions but it depends on the action functions $f()$ and $g()$. These action functions might never return 1 and 2 at the same time, but this information is not available to the Verifier.

When verifying in basic mode, all guard expressions are assumed to be true, and the Verifier will report conflicting transitions in all three Systems. When verifying in guard mode, the Verifier will report conflicting transitions in Systems b and c. When verifying in full mode, the Verifier will only report conflicting transitions for the System c.

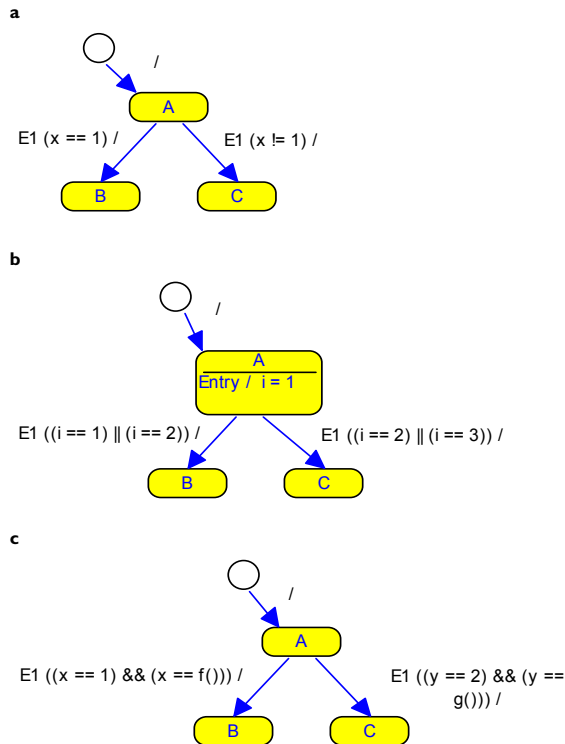


Figure 96: Systems with conflicting transitions.

All three Systems have conflicts that must be resolved manually when verifying in basic mode. When verifying in guard mode no conflicting transitions are reported for the System a. When verifying in full mode, only System c has conflicting transitions that must be resolved manually.

Checks performed by visualSTATE Verifier

This chapter gives a detailed description of the checks that the Verifier can perform, illustrated with examples. The chapter also lists the modes in which the various Verifier checks can be performed, and describes how to interpret verification warnings and error messages.

For information on how to activate the various Verifier checks, see *Verifying your visualSTATE Project*, page 137, or *Verifier command line options*, page 371.

Check for unused elements

The Verifier performs a static analysis of a visualSTATE System to check if all declared elements are used. The following elements are checked:

- States
- Variables, event parameters, and constants
- Action functions
- Events, event groups, and signals.

STATES

A state is reported as unused if it is neither the source state or destination state of any transition, nor the default state of an initial state, shallow history state, or deep history state.

VARIABLES, EVENT PARAMETERS, AND CONSTANTS

Variables are said to be read if they are used in guard expressions, or the right-hand side of an assignment, or as parameters to action functions. They are said to be written if used on the left-hand side of an assignment.

External variables are reported as unused if they are neither read nor written on any transitions or state reactions.

Internal variables are reported as statically unread if they are not read on any transitions or state reactions.

Internal variables are reported as statically unwritten if they are not written on any assignments or state reactions.

Event parameters and constants are reported as unused if they are not read on any transitions or state reactions.

ACTION FUNCTIONS

Action functions that are not used on any transitions or state reactions are reported as unused.

EVENTS, EVENT GROUPS, AND SIGNALS

Events and event groups that are not used as triggers for any transitions or state reactions are reported as unused.

Signals on transitions or state reactions that are never sent are reported as never sent.

Signals that are not used as triggers for any transitions or state reactions are reported as never used as triggers.

Example

The System in *Figure 97*, page 124, has the following elements defined:

Events: `E1 (VS_INT par0) , E2`
 Internal variable: `i`
 External variable: `x`
 Signal: `S1`

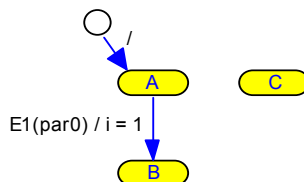


Figure 97: System with unused elements

Performing a Vericator check on the System shown in *Figure 97*, page 124, gives the following result for unused elements:

Unused states:
 C

Never read internal variables (static check):
 i

Unused external variables:

x

Unused event parameters:

E1.par0

Unused events:

E2

Signals never used as triggers (static check):

S1

Never sent signals (static check):

S1

Check for activation of elements

The check for activation of elements is similar to the check for unused elements (see *Check for unused elements*, page 123) but it is based on the *dynamic* behavior of the System. The static verification check is similar to the syntax check of a compiler, whereas the dynamic check analyzes the behavior of the running visualSTATE System.

In the following, a transition is said to be *reachable* if a sequence of events can lead to the transition being triggered.

The following elements are checked for activation:

- States
- Variables, event parameters, and constants
- Action functions
- Events, event groups, and signals
- Transitions.

STATES

A state is reported as never activated if it is not part of a reachable state configuration.

VARIABLES, EVENT PARAMETERS, AND CONSTANTS

A transition's guard expressions are considered activated if the source state of the transition is reachable.

A transition's assignments and action functions are considered activated if the source state of the transition can be reached and the transition can be triggered.

External variables are reported as never activated if they are neither read nor written in any activated guard expression or assignment, or used as parameter for any activated action function.

Internal variables are reported as dynamically unread if they are not read in any activated guard expression or assignment, or used as parameter for any activated action function.

Internal variables are reported as dynamically unwritten if they are not written in any activated assignment.

Event parameters and constants are reported as never activated if they are not read in any activated guard expression or assignment, or used as parameter for any activated action function.

ACTION FUNCTIONS

When action functions returning values (`non-void` functions) are used in guard expressions and assignments, they are treated as event parameters and constants.

When action functions are used outside guard expressions or assignments, they are considered activated if the transitions on which they are used are reachable.

EVENTS, EVENT GROUPS, AND SIGNALS

Events and event groups that are not used as triggers for any reachable transition are reported as never activated.

Signals that are not used on the transition action side of any reachable transition are reported as never sent.

Signals that are not used as triggers for any reachable transitions are reported as never used as triggers.

TRANSITIONS

Transitions that can never be triggered are reported as never activated.

Example

The System in *Figure 98*, page 127 has the following elements defined:

Events: `E1(VS_INT par0), E2, E3`

Internal variable: `i`

External variable: `x`

Signals: S1

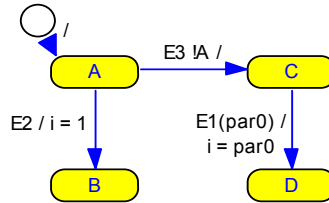


Figure 98: System with never activated elements

Performing a Verificator check on the System shown in *Figure 98*, page 127 gives the following result for never activated elements:

Unactivated states:

C, D

Never read internal variables (dynamic check):

i

Unactivated external variables:

x

Unactivated event parameters:

E1.par0

Unactivated events:

E1, E3

Signals that never act as triggers:

S1

Never sent signals (dynamic check):

S1

Unactivated transitions:

C:

E1(par0) / [i = par0]

-> D

A:

E3() !A /

-> C

Check for conflicting transitions

Two transitions with common trigger and source state, but different destination states are said to be conflicting if they both can be triggered at the same time. It is an error if a System has conflicting transitions.

Example

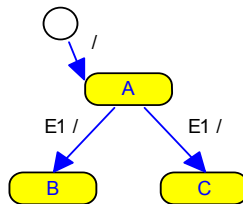


Figure 99: System with conflicting transitions

Performing a Vericator check on the System shown in *Figure 99*, page 128 reports the following results for conflicting transitions:

The following transitions conflict:

```
A:
  E1 () /
-> B
```

```
A:
  E1 () /
-> C
```

Check for state dead ends

A state dead end is a state in a state machine that once entered cannot be left.

Example

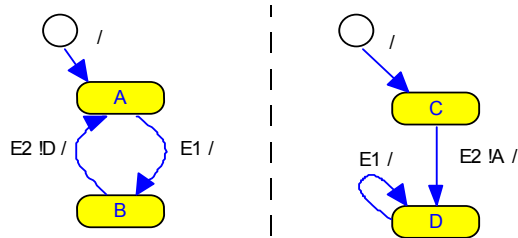


Figure 100: System containing a state dead end.

The System consists of two state machines. State B in the left-hand state machine is not a state dead end, although it cannot be left after it has been entered for the second time. State D in the right-hand state machine is a state dead end because the state machine cannot change state after state D has been entered for the first time.

Performing a Verificator check on the System shown in Figure 100, page 129, reports the following state dead end result:

State dead ends

D

Here, no sequence of events can make the second state machine leave state D after it has been entered for the first time.

Note: This check is not performed in compositional mode.

Check for local dead ends

A local dead end in a state machine M is a set of states that makes M unable to change state.

Example

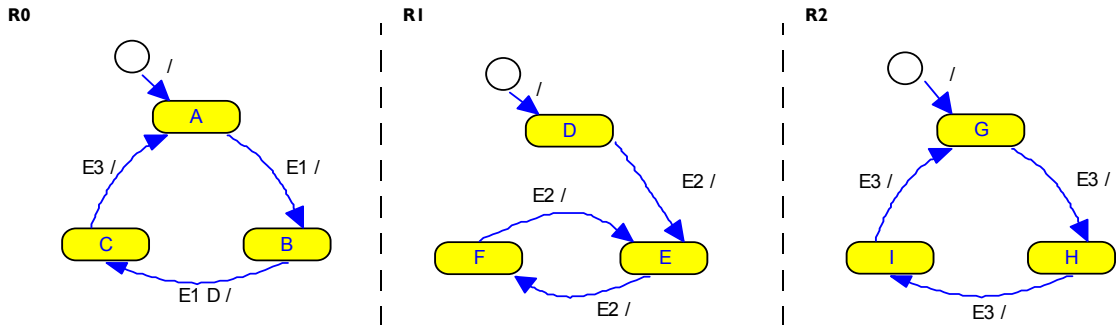


Figure 101: System containing a local dead end.

The System consists of three state machines. The first machine deadlocks when the System enters the state configurations (B, F) and (B, E).

Performing a Verificator check on the System shown in *Figure 101*, page 130, gives the following local dead end result:

Local dead end for the machine: R0

$\{B\} \times \{E, F\} \times \{*\}$

The local dead end above can be reached by the event sequence E1, E2.

Check for System dead ends

A System dead end is a state configuration that renders all state machines in the System deadlocked.

Example

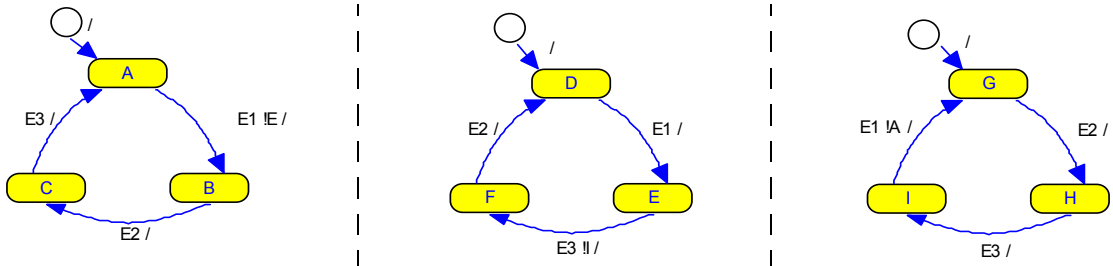


Figure 102: System containing a System dead end.

The System consists of three state machines. The System can reach the state configuration (A, E, I) which is a System dead end.

Performing a Verificator check on the System shown in Figure 102, page 131 gives the following result for System dead ends:

System dead ends

$\{A\} \times \{E\} \times \{I\}$

The System dead end above can be reached by the event sequence E2, E3, E1, E2, E3.

Note: This check is not performed in compositional mode.

Check for dynamic ambiguous assignments

Systems should not execute multiple simultaneous assignments or simultaneously assign and read the same variable. The reason is that multiple triggered transitions should be considered as either being triggered at the same time, or being triggered in an unspecified sequence.

Example

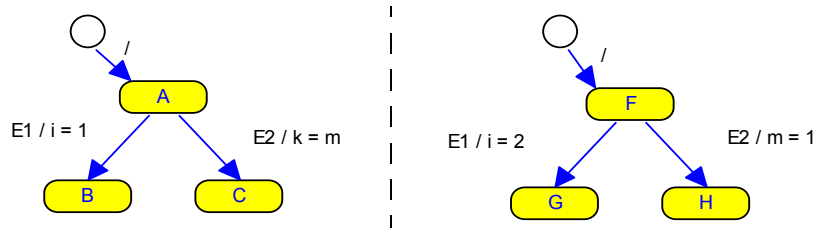


Figure 103: System containing dynamic ambiguous assignments.
 The System consists of two state machines. The event E1 will trigger the two transitions which both assign *i* making the value of *i* ambiguous. The event E2 will trigger two transitions, one reading *m* (A -> C) and one assigning *m* (F -> H) making the value of *k* ambiguous.

Performing a Vericator check on the System shown in Figure 103, page 132, gives the following ambiguity results:

The variable *i* is assigned several times on the transitions

A:

E1 () / [i = 1]

-> B

and

F:

E1 () / [i = 2]

-> G

The variable *m* is both assigned and read on the transitions

A:

E2 () / [k = m]

-> C

and

F:

E2 () / [m = 1]

-> H

Note: This check is only performed in full mode.

Check for static ambiguous assignments

When there are multiple assignments on a single transition, they are executed in some fixed sequence in the code generated by the visualSTATE Coder. However, such assignments cannot be handled in a full mode verification if they involve the same variable in more than one assignment expression. Likewise, multiple ambiguous assignments on a single transition should be avoided if you want to verify your System in full mode.

Example

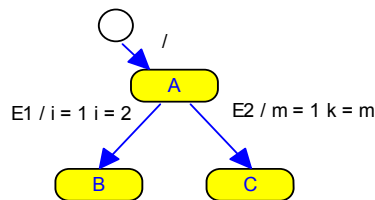


Figure 104: System with two transitions having ambiguous assignments.
The transition A -> B assigns i twice. The transition A -> C both reads and writes m.

Performing a Verificator check on the System shown in *Figure 104*, page 133 gives the following static ambiguity results:

The variable i is assigned several times on the transition A:

```
E1 () / [i = 1] [i = 2]
-> B
```

The variable m is both assigned and read on the transition A:

```
E2 () / [m = 1] [k = m]
-> C
```

Note: This check is only performed in full mode.

Check for signal queue size

When signals are used, a signal queue size must be specified. Do not specify a larger signal queue than necessary, because the complexity of verifying the model greatly increases with increased size of the signal queue. If the queue is too large, a minimum required size is reported. If the queue is too small, the Verificator will report queue

overflow, unless the drop-if-full signal queue option is selected in the Designer (see *Specifying signal queue behavior*, page 95). Signal queue overflow is an error which means that the remaining part of the verification will be based on false assumptions.

For an example of a System requiring a signal queue size and type, see *Figure 105*, page 134. For additional information about signals and signal queues, see *IAR visualSTATE Reference Guide*.

Example

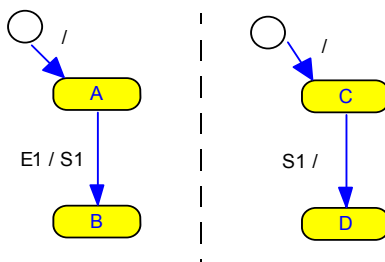
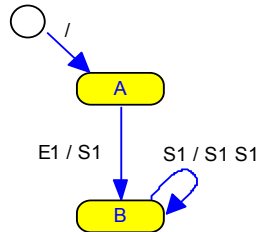


Figure 105: System for which the size of the signal queue must be at least one. The System consists of two state machines.

Performing a Vericator check on the System shown in *Figure 105*, page 134 gives the following signal queue results:

- If a signal queue of length 0 is specified:
The signal queue is too small.
- If a signal queue of length 1 is specified:
The signal queue has the right size.
- If a signal queue of length 2 is specified:
The signal queue is too large. Only 1 element is needed in the queue

Note: Systems that need an unbounded signal queue cannot be fully verified. *Figure 106*, page 135 shows an example of such a System.



*Figure 106: System which cannot be fully verified.
The System will continue adding signals to the signal queue until the queue overflows, resulting in incorrect verification.*

Overview of checks, modes, and errors

Table 4, page 136 lists the Verificator checks performed in the various modes, and whether the errors given in Verificator check reports should be considered critical errors. If a critical error occurs, the System contains logical errors and it may not be possible to verify it. You are recommended not to code-generate Systems containing critical errors.

Check	Check performed in				Critical error
	Basic mode	Guard mode	Full mode	Comp. mode	
Unused elements					
States	Yes	Yes	Yes	Yes	No
Variables, event parameters, and constants	Yes	Yes	Yes	Yes	No
Action functions	Yes	Yes	Yes	Yes	No
Events, event groups, and signals	Yes	Yes	Yes	Yes	No
Activation of elements					
States	Yes	Yes	Yes	Yes	No
Variables, event parameters, and constants	Yes	Yes	Yes	Yes	No
Action functions	Yes	Yes	Yes	Yes	No
Events, event groups, and signals	Yes	Yes	Yes	Yes	No
Transitions	Yes	Yes	Yes	Yes	No
Conflicting transitions	Yes	Yes	Yes	Yes	Yes
State dead ends	Yes	Yes	Yes	No	No
Local dead ends	Yes	Yes	Yes	Yes	No
System dead ends	Yes	Yes	Yes	No	No
Dynamic ambiguous assignments	No	No	Yes	No	Yes
Static ambiguous assignments	No	No	Yes	No	Yes
Signal queue	Yes	Yes	Yes	Yes	Yes ^a

Table 4: Verificator checks, modes and errors

a. Unless a drop-if-full signal queue is specified in the design.

Verifying your visualSTATE Project

This chapter describes how to start verification from the Navigator.

The verification process can also be started from the command line. For a description of using the command line method, see *Verificator command line options*, page 371.

For a general description of formal verification and Verificator checks, see *Introduction*, page 109, and *Checks performed by visualSTATE Verificator*, page 123.

Starting verification

- 1 Open the workspace file that contains the visualSTATE System you want to verify.
- 2 Ensure that the appropriate Verificator options have been set for the active System. Choose Project>Options>Verification... to open the Verificator Options dialog box. See *Figure 107*, page 137.

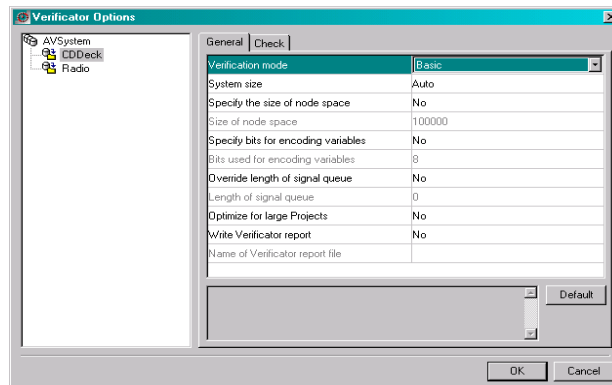


Figure 107: Verificator Options dialog box, General tab

Click the General tab to set the appropriate general Verificator options.

Click the Check tab to select the checks to be performed.

In the tree browser to the left, select other Systems for which to set options.

Note: Not all combinations of options are possible because the values selected for one option may limit the choices for other options. This is described in the online help for the option (see *Online help*, page 31).

For a general description of how to set options, see *Setting Verificator, Coder and Documenter options*, page 29.

- 3 On the Project menu, choose *Verify System* or *Verify All Systems*, whichever is relevant.

If there is more than one System in the Project, and you choose *Verify All Systems*, a dialog box will be displayed where you can select the System(s) to verify. See *Figure 108*, page 138.

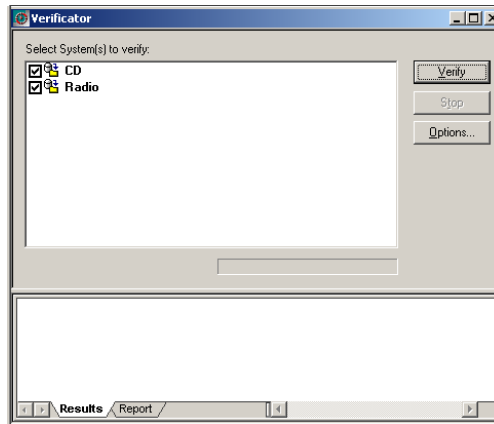


Figure 108: Verificator dialog box

Select the appropriate System(s), and click the *Verify* button.

- 4 If your model has been changed, the message shown in *Figure 109*, page 139 will be displayed, unless you have turned off the option *Show notification if model has*

changed (see *visualSTATE Project code-generated via the Navigator*, page 34). If this option is deselected, no warning is given, and code generation is not performed.

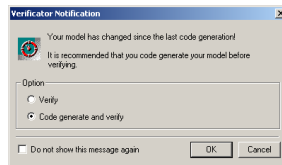


Figure 109: Vericator notification

- 5 A verification progress window will be opened, and the selected item(s) will be verified. The items are listed by groups of checks. See *Figure 110*, page 139.

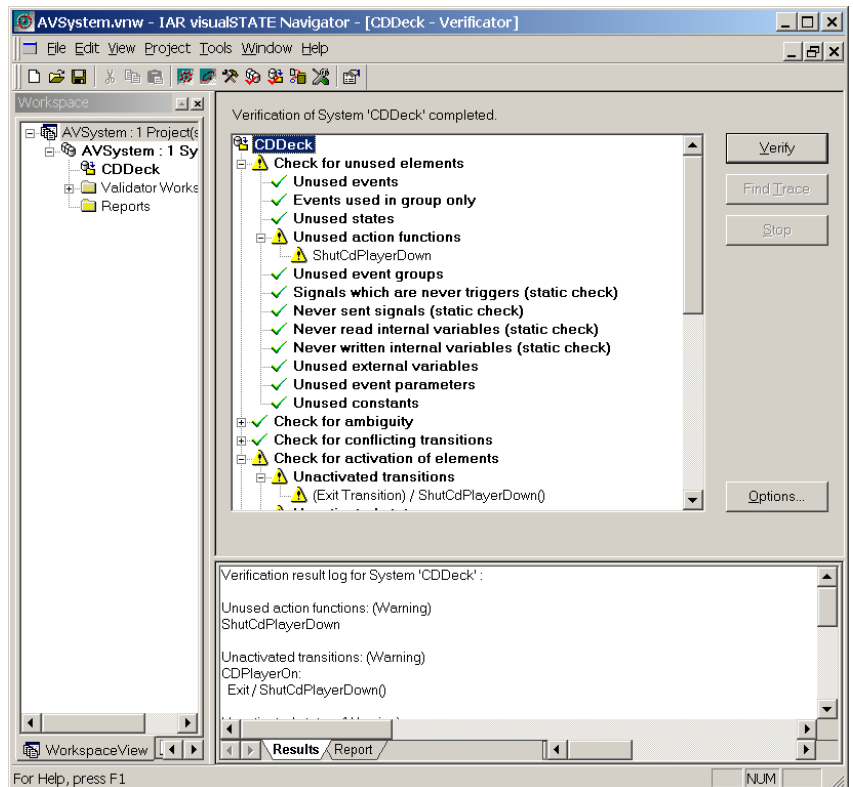


Figure 110: Verification progress window, Navigator

The verification progress window gives you an immediate view of the results of the verification. Items selected for verification are shown in bold (in the upper part of the progress window). Items that have been verified but have caused errors or warnings are expanded and marked.

To see the cause of a warning or an error, select the item in the upper part of the verification progress window and see the description displayed in the lower part (Results tab). To view the results for an entire System, select the System in the upper part of the window.

In some cases you can get a trace to the error or warning. Refer to *Tracing your visualSTATE Project*, page 141 for more information on that.

To change Vericator options for a System, select the System in the tree browser in the upper part of the window, and click the **Options** button to open the Vericator Options dialog box. To perform another verification, click the **Verify** button.

If you selected **Yes** for **Write Vericator report** in the Vericator options dialog box (see *Figure 107*, page 137), you can view a summary of the completed verification by clicking the Report tab in the verification progress window (see *Figure 110*, page 139).

Tracing your visualSTATE Project

This chapter describes how to trace by means of the Verificator from the Navigator.

A trace is a sequence of events that will get the System into a desired state configuration. The trace will be saved in a test sequence file. To read more on test sequence files refer to *Playing recorded test sequences*, page 191.

The tracing process can also be started from the command line. For a description of using the command line method, see *Verificator command line options*, page 371.

For a general description of formal verification and Verificator checks, see *Introduction*, page 109, and *Checks performed by visualSTATE Verificator*, page 123.

Performing a trace

- 1 You can only perform a trace in the Navigator if you have just run a verification. If you have not just done that refer to *Starting verification*, page 137.
- 2 Mark the dead end or conflict you want to trace to. When you have marked one the **Find Trace** button will be enabled. Refer to *Figure 111*, page 141.

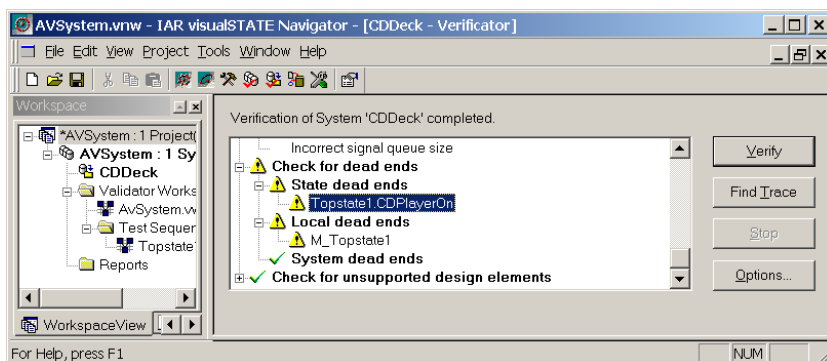


Figure 111: Verificator Results, Ready to Find Trace

- 3 Click the **Find Trace** button.

- 4 Select or specify the file name for the trace output file and click **Save**. See *Figure 112*, page 142.

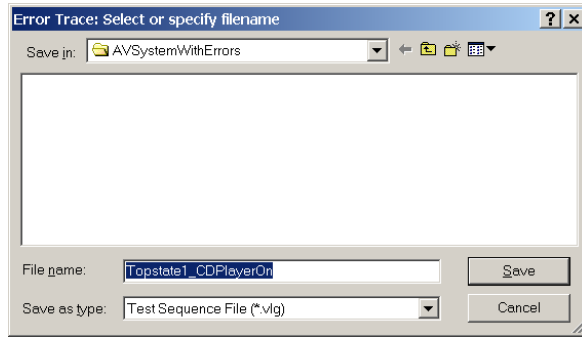


Figure 112: Specifying trace output file name

- 5 After that the Navigator will find a trace to the error or warning and save the resulting trace. After the file has been saved the Validator will be opened with the test sequence file loaded. Refer to *Recording and playing test sequences*, page 187.

Designing for verification

This chapter describes time/memory managing options, gives recommendations on modeling, and lists the constructs that should preferably be avoided if you want to be able to verify your visualSTATE Systems.

Some constructs might make a visualSTATE System too complex to verify in a reasonable amount of time or memory, whereas other constructs should be avoided to ensure that the behavior of the generated code will exactly match the verification result.

Using time/memory options to help verification

Generally the use of time and memory by the Verificator grow with the number of states, transitions and signal queue size of the visualSTATE System. A computer always have limited memory, so at one point the memory requirement will be the deciding factor. To some extent it is possible to trade time for memory by using advanced techniques.

SMALL/LARGE OPTIONS

As a default the Verificator tries to decide whether a visualSTATE System would benefit from minimizing the time usage or the memory usage. A small System does not demand large amounts of memory, and the Verificator focuses on minimizing the time usage. Large Systems use large amounts of memory, which can lead to bottleneck problems such as disk swapping, so it will be beneficial to minimize the memory usage. In some cases this will even be quicker than having the time minimizing option set, or the only possibility for achieving a successful verification.

The user can control the two techniques with these two options:

- **-Small** or **-small** option for minimizing time usage.
- **-Large** or **-large** option for minimizing memory usage.

NODE SPACE SIZE OPTION

Node space is the memory area used for the data structures build during a verification. It is impossible beforehand to find the necessary size of the node space, so the right size must be found by trial and error. If the node space is too large the Verificator is tying up valuable resources. If the node space is too small the node space is automatically expanded in an exponential fashion, which can lead to memory fragmentation. Normally the Verificator can handle the node space requirement itself, but for large visualSTATE

Systems it can be beneficial to set the initial size of the node space by hand. This is done with the **S** option:

- **-S<n>**, where **n** is the initial size of the node space.

The size of the node space is measured in nodes. Each nodes occupies 20 bytes, but some overhead and caching makes the real memory requirement per node approximately 30-35 bytes.

Keeping down the complexity of verifying Systems

It is possible to design visualSTATE Systems that are so complex that they cannot be verified in a reasonable amount of time or memory. Therefore, you are recommended to consider the following to keep down the complexity of verifying your Systems, and thereby reduce time consumption:

- Verification mode
- Signals and signal queue
- Operators
- Depth of state space.

VERIFICATION MODE

Full mode verification is more complex than guard mode verification which again is more complex than basic mode verification.

Compositional mode verification can often handle very large Systems, but is most suitable for Systems consisting of many independent state machines. State machines are independent if they do not use signals and only use state conditions sparingly.

For detailed information about verification modes, see *Verification modes*, page 114.

SIGNALS AND SIGNAL QUEUE

In all verification modes, the use of signals and the size of the signal queue influence the complexity of verification. The signal queue should be kept as small as possible, and it should not overflow. For additional information, see *Check for signal queue size*, page 133.

For detailed information about signal queue, see *Check for signal queue size*, page 133.

OPERATORS

The following guidelines apply to the use of operators:

- Do not use the following operators with variables larger than 8 bits:
*, /, %, <<, >>.
- In full mode and guard mode, the bit size of variables that is *actually used* should be as small as possible. For example, avoid representing a number of binary flag values in a 32-bit variable—use separate `VS_BOOL` variables instead.
- Do not use a `VS_UINT32` if a `VS_BOOL` is sufficient.
- Use simple expressions with few arithmetics operators.
- If the native integer size of your target MCU is 16 bits, indicate the integer size to the Vericator by specifying the 16-bit `int` option.
- Specifying that all variables should be encoded using some small number of bits might make it possible to verify an otherwise too complex System. Use this method with care, because it often changes the semantic meaning of the model radically.

For detailed information, see *Variables, domains, and arithmetics*, page 121.

DEPTH OF SYSTEM STATE SPACE

Avoid Systems with System configurations which can only be reached after very long event sequences. The Systems in *Figure 113*, page 145 and *Figure 114*, page 145 have the same state space, but the number of events needed to reach any System configuration is much larger in the first System than in the second System.

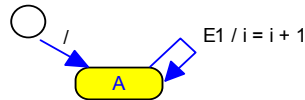


Figure 113: System with deep state space.

i is an internal variable of type `VS_UINT32` initialized to 0. Verifying this System in full mode will take some time because the only event sequence leading to the System configuration (A, $i = 2^{32}-1$) is the sequence consisting of $2^{32}-2$ times the event E1.

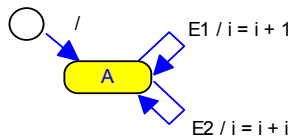


Figure 114: System with shallow state space.

i is an internal variable of type `VS_UINT32` initialized to 0. This System has the same state space as the System in *Figure 113*, page 145, but any of the System configurations can be reached in 65 steps or less.

Verification and visualSTATE generated code

The following factors influence the verification in relation to visualSTATE generated code:

- Expressions
- Environment
- Non-verifiable elements.

EXPRESSIONS

The Verificator evaluates expressions in the domain of the variables in the expression. If for example the type of the variables in the expression is `VS_UINT8`, the arithmetic is performed using eight bits. This is done to keep down the complexity of the verification but a few precautions should be taken. C evaluates expressions using `int` or `long int` arithmetic. This might make a difference when an expression or subexpression wraps around.

For detailed information, see *Variables, domains, and arithmetics*, page 121.

ENVIRONMENT

When verifying in full mode the model is placed in an environment where any sequence of events is possible and all external variables can take on any value. The actual environment in which the final product is used is a proper subset of the Verificator environment. This ensures that any potential inconsistencies are detected, as well as inconsistencies that might not show up in the final product.

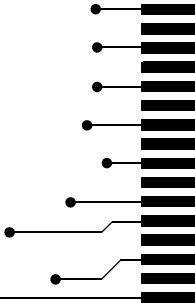
NON-VERIFIABLE ELEMENTS

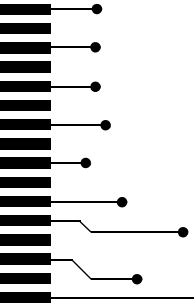
For a list of non-verifiable elements, see *Non-verifiable elements*, page 119.

Part 5: Functional testing

This part of the visualSTATE[®] User Guide includes the following chapters:

- Introduction
- Simulation
- Tracing
- Recording and playing test sequences
- Analyzing visualSTATE models.





Introduction

Functional testing in visualSTATE is performed by means of the Validator which has tools for simulating, analyzing, tracing, and debugging visualSTATE models created with the Designer.

This chapter gives an introduction to simulation with visualSTATE Validator and describes the graphical environment of the Validator, including toolbars.

Simulation with visualSTATE Validator

Simulation with visualSTATE Validator is characterized by the following, in contrast to verification with visualSTATE Verificator (see *Verification with visualSTATE Verificator*, page 110):

- Functionality test. Checks that the application is in accordance with your requirements specification.
- Gives you insight in the behavior of the model at specific points of execution.
- Test your visualSTATE model in a target application by means of RealLink (see *Part 6: Testing in target applications*, page 201).
- Test is based on user interaction.

VALIDATOR TOOLS

The Validator has tools for:

- Interactive simulation, including graphical animation and use of conditional breakpoints (see *Simulation*, page 161).
- Tracing - i.e. to get a sequence of events that will get the System into a desired configuration (see *Tracing visualSTATE models*, page 183).
- Automatic simulation, by recording and playing test sequences by means of test sequence files (see *Recording and playing test sequences*, page 187).
- Listing the visualSTATE elements used, and test coverage (see *Analyzing visualSTATE models*, page 195).

SIMULATION MODES

The Validator has two simulation modes:

Validator mode In this mode, you simulate your visualSTATE model.

Target mode

In this mode, you can monitor and control your visualSTATE model in a target application by means of the Validator RealLink facility.

For information on how to change between the modes, see *Toggleing between Validator mode and target mode*, page 180.

Graphical environment

The Validator environment consists of a number of windows with pop-up menus, menus, and toolbars. See example in *Figure 115*, page 150.

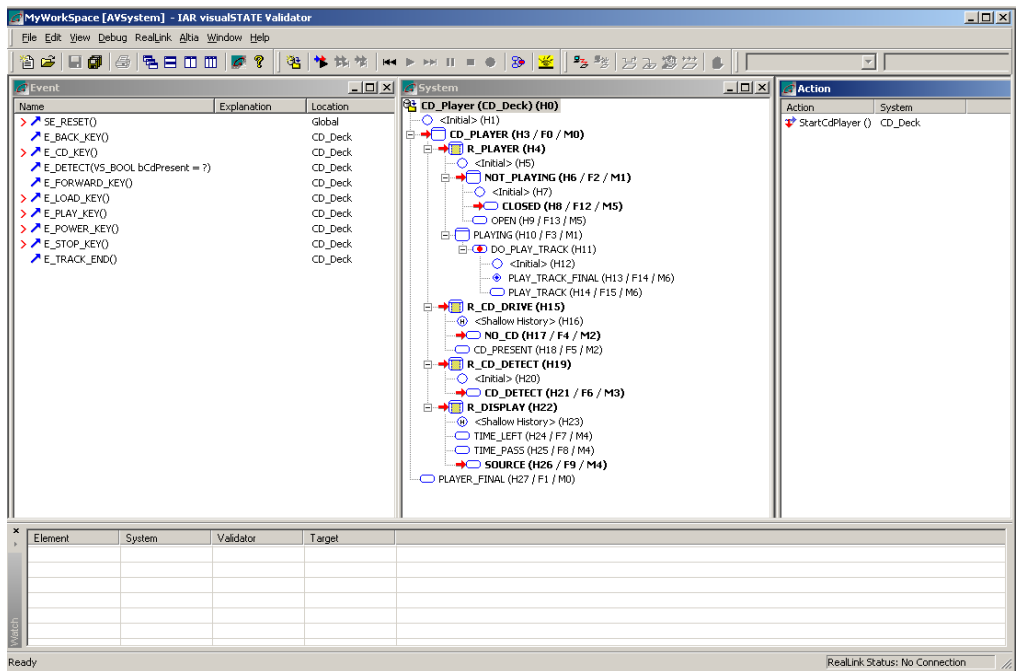


Figure 115: Validator environment with workspace loaded

THE VALIDATOR WORKSPACE

The Validator workspace is a file containing information on your validation session (file name extension is `vws`).

The workspace file saves information about which Project is loaded, the setup of the current test session, including breakpoints, and window setup. You are recommended always to save the setup of your test session in a workspace.

Note: It is only possible to have one visualSTATE Project in a workspace.

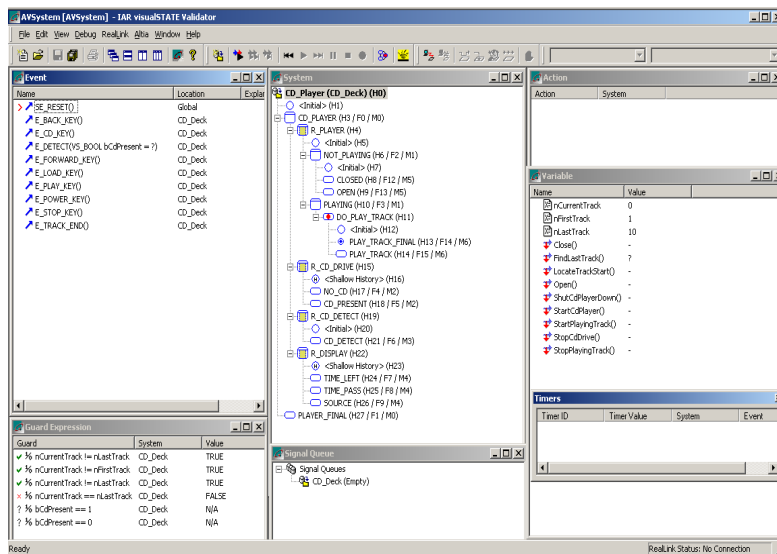


Figure 116: Validator workspace, customized window setup

It is possible to have more than one Validator workspace, each loading the same visualSTATE Project, and each having its own particular setup. This is useful when testing different aspects of a visualSTATE Project.

When you launch the Validator from the Navigator you will automatically get an appropriate workspace for the Project in the Validator.

Creating a new Validator workspace

- 1 On the Validator menu, choose File>New Workspace. A dialog box will be opened. See *Figure 117*, page 152.

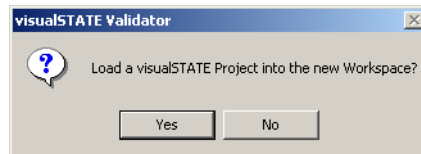


Figure 117: New Validator workspace dialog box

- 2 Click **Yes**. An Open Project dialog box will be opened.
- 3 In the dialog box, specify the visualSTATE Project to load. The selected Project will be opened in the workspace.
- 4 Choose File>Save, and save the workspace using the Save As dialog box displayed.

Note: Do not change the `vws` extension of the Validator workspace file.

The Project will be loaded into the workspace. See example in *Figure 115*, page 150.

Opening and closing a workspace

To open an existing workspace, choose File>Open. In the Open Validator Workspace dialog box displayed, select the workspace file to open.

To close a workspace, choose File>Close Workspace.

VALIDATOR WINDOWS

The Validator has a number of windows that provide information on the various aspects of a visualSTATE Project. All windows have pop-up menus by which you can activate various commands. The Validator windows are opened via the Windows menu and the View menu. For the latter type of window, you can use the shortcuts ALT + *number* to set focus to windows that are already open.

System window

To open this window, choose Windows>New Window on the menu.

This window gives a hierarchical view of the Systems designed in the Project.

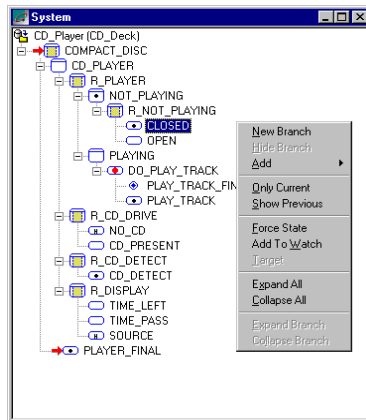


Figure 118: System window (Validator), with pop-up menu

The default setup of the System window shows each System and each of their instances in a separate branch in the tree.

Event window

To open this window, choose Windows>New Window on the menu.

The Event window provides a view of all events defined in the loaded Project, and is used for sending events into the System(s). See *Figure 119*, page 153.

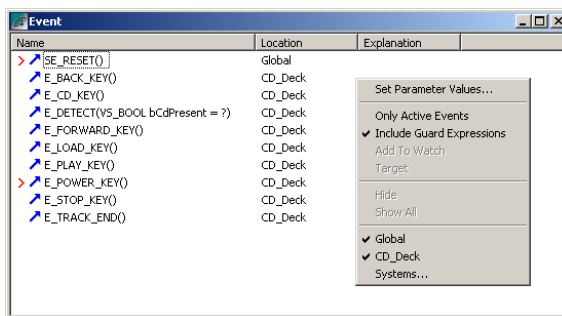


Figure 119: Event window (Validator), with pop-up menu

The window provides information about the following:

- Event name.

- Which events are active (an active event is an event, which, if sent, will trigger one or more transitions).
- Explanation for events.
- Location of event definition.

Action window

To open this window, choose Windows>New Window on the menu. See example in *Figure 120*, page 154.

The window shows what happened during the last deduction, and provides information about the following:

- Which variables were assigned a value.
- The value assigned to the variables.
- Which actions were executed, and the arguments with which they were called.

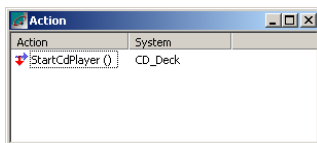


Figure 120: Action window (Validator)

Variable window

To open this window, choose Windows>New Window on the menu.

The Variable window shows all variables, all action functions and all constants declared in all Systems. See *Figure 121*, page 155.

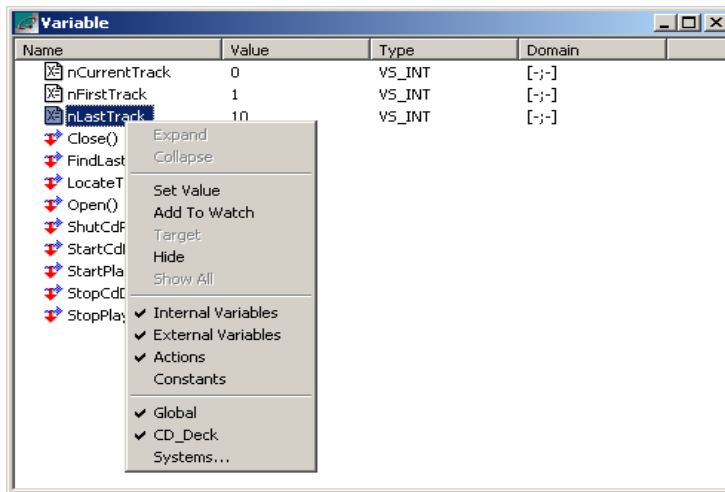


Figure 121: Variable window (Validator), with pop-up menu

Guard Expression window

To open this window, choose Windows>New Window on the menu.

The Guard Expression window gives an overview of all guard expressions defined in all Systems (see *Figure 122*, page 155).

Guard	Value	System
✓ % nCurrentTrack != nLastTrack	TRUE	CD_Deck
✓ % nCurrentTrack != nFirstTrack	TRUE	CD_Deck
✓ % nCurrentTrack != nLastTrack	TRUE	CD_Deck
✗ % nCurrentTrack == nLastTrack	FALSE	CD_Deck
? % bCdPresent == 1	N/A	CD_Deck
? % bCdPresent == 0	N/A	CD_Deck

Figure 122: Guard Expression window (Validator)

Signal Queue window

To open this window, choose Windows>New Window on the menu.

The Signal Queue window is used for signal queue handling, and provides a view of the signal queues in all Systems and instances (see *Figure 123*, page 156).

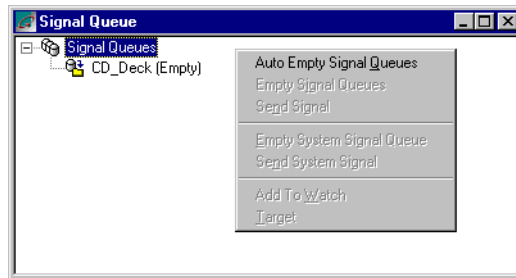


Figure 123: Signal Queue window (Validator), with pop-up menu

Field Chooser window

This window is used for displaying or hiding columns in the Event window, Action window, Variable window, and Guard Expression window. The Field Chooser window displays the columns in the currently active window. See example in *Figure 124*, page 156.

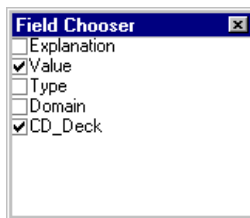


Figure 124: Field Chooser window for Variable window (Validator)

In this example, clicking the Explanation check box will display the Explanation column in the Variable window. Clicking the Value check box will hide the Value column in the Variable window.

System Setup window

This window is used for setting up the order in which visualSTATE Systems should be simulated. See *System setup*, page 178.

Output window

This window provides various save and load information and has the following views (click the tabs to change view):

General	Shows information during load.
RealLink	Shows messages related to the latest RealLink connection.
Altia	Shows messages related to the latest Altia connection.

See *Figure 125*, page 157.

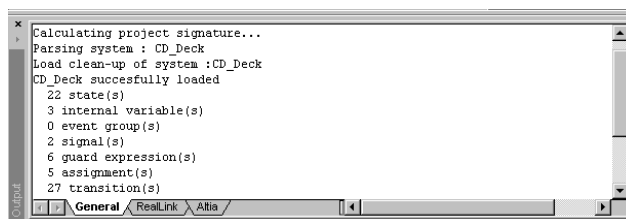


Figure 125: Validator output window

Watch window

In the Watch window you can add any type of element contained in the visualSTATE model. Thus in the Watch window you can have a collection of the elements that you find most interesting. The elements are added to the Watch window from the System, Event, Variable, and Signal Queue windows.

To add an element to the Watch window:

- 1 Open the window containing the element to add, for example the Event window.
- 2 Open the pop-up menu, and choose **Add to Watch** (see example in *Figure 119*, page 153), or press SHIFT+F9. The element will be added to the Watch window. See *Figure 126*, page 157.

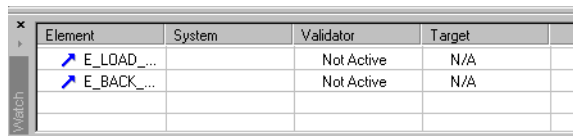


Figure 126: Validator Watch window with elements added

Timers window

This window displays the values of all running timers.

You can stop timers via the pop-up menu of the window (see *Figure 127*, page 158), or by pressing DELETE.

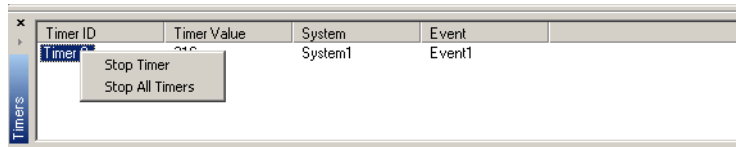


Figure 127: Validator Timers window, with pop-up menu

Breakpoints window

This window shows all defined breakpoints. Furthermore it is possible to disable a breakpoint in this window. See *Breakpoints*, page 169.

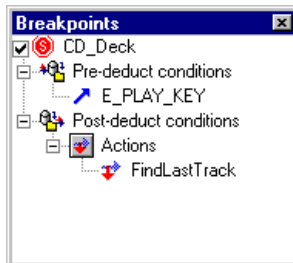


Figure 128: Validator Breakpoints window

VALIDATOR TOOLBARS

The most frequently used menu commands are available as toolbar buttons with tooltips. A detailed description of the Validator menu commands is found in *Validator menu commands*, page 357).

The following toolbars are available:

- Standard toolbar, see *Figure 129*, page 159.
- Debug toolbar, see *Figure 130*, page 159.
- RealLink toolbar, see *Figure 131*, page 159.
- Analysis toolbar, see *Figure 132*, page 159.

Simulation

This chapter describes how to use the visualSTATE Validator for interactive simulation of visualSTATE models created with the Designer, and how you can view your interactive simulation graphically in the visualSTATE Designer.

By interactive simulation you manually send events into one or more visualSTATE Systems and view the System's reaction to this, including variables assigned a new value, generated signals, actions, and state changes in the System(s) simulated.

You can also simulate your visualSTATE model automatically by applying commands that have been recorded to a file. This is described in *Recording and playing test sequences*, page 187.

Starting simulation

Before you start simulation and send events into the System, you must:

- Initialize the loaded Systems
- Send the reset event
- Set event parameters.

This is done as follows:

- 1** Launch the Validator from the Navigator, to open the Validator workspace containing the Project that you want to simulate.
- 2** *On the menu, choose Window>Classic Simulation.*
- 3** *On the Debug toolbar, click the Initialize button to initialize the System(s).*

If the Project contains more than one System/instance, the command will display a dialog box for selecting the Systems to initialize. See *Figure 133*, page 162.

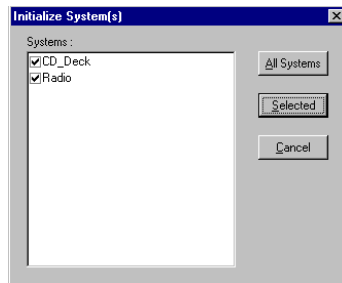


Figure 133: Initialize Systems dialog box (Validator)

- 4 In the Event window, double-click SE_RESET to send the visualSTATE reset event into the System(s). Active events are marked by a red arrowhead. See *Active events*, page 164.

Note: The reset event name is always SE_RESET and cannot be changed.

You are now ready to start simulation by sending events into the loaded System(s).

See *Sending events*, page 162.

Sending events

When you have completed the steps described in *Starting simulation*, page 161, you can send events into the loaded System(s), as follows:

In the Event window, double-click on the event that is to be sent.

Note: If the event has any parameters, they should all be assigned a value before the event is sent (see *Specifying event parameters*, page 167). If not, it is not allowed to send the event. The reason is that the event parameters may be used in a guard expression or an assignment, and it is not possible to resolve these without having the value of the event parameters. In contrast, it is allowed to send an inactive event but it will produce a warning, and it will not cause any reaction from the System.

Global events will be sent to all enabled visualSTATE Systems. Local events will be sent to the System in which they are defined. For enabling and disabling of Systems, see *System setup*, page 178.

When you have sent the event, new events may become active, shown by a red arrowhead. See example in *Figure 134*, page 163.

For sending events you can also use the Watch window. See *Using the Watch window for sending events*, page 164.

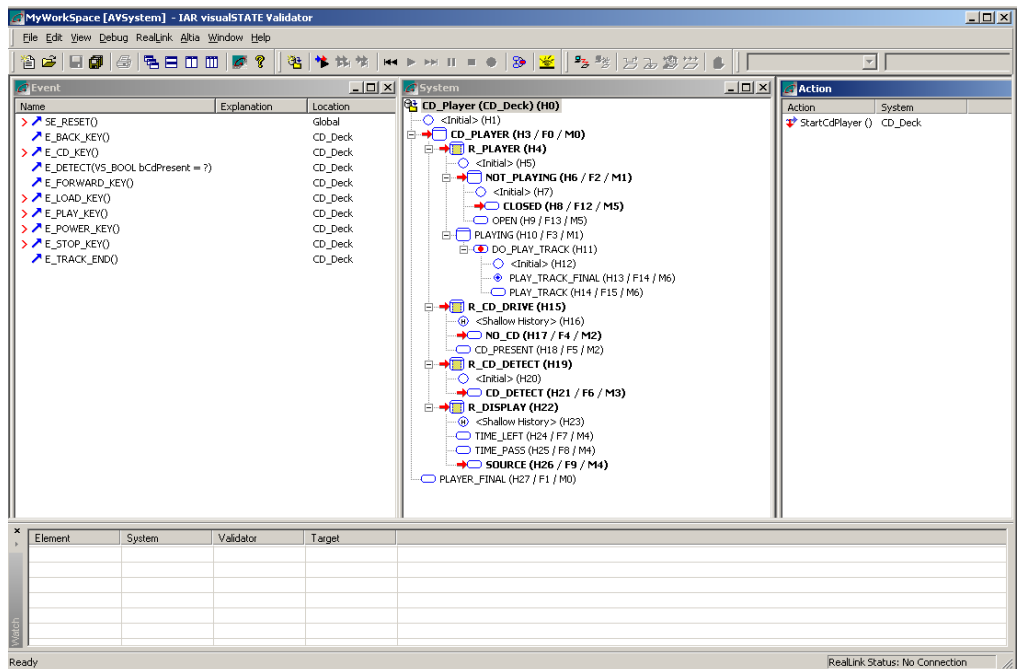


Figure 134: Validator environment with workspace loaded

You can filter the information in the Event window as follows:

To hide event(s) from the Event window, select one or more events, and choose **Hide** on the pop-up menu. It is possible to hide events declared in a particular System. All Systems are listed at the bottom of the pop-up menu.

To display all hidden events, choose **Show All** on the Event window pop-up menu.

Events can also be viewed in the Watch window. See *Watch window*, page 157.

To have guard expressions resolved during the inquiry on active events, choose **Include Guard Expressions** on the Event window pop-up menu.

When guard expressions are included, only guard expressions evaluated as FALSE will make an event inactive. Guard expressions evaluated as TRUE, and expressions that cannot be evaluated (marked "N/A" in the Guard Expression window) will not cause an event to be inactive.

Note: The *Include Guard Expressions* option is only available in Validator mode (not RealLink) because the inquiry on active events by the visualSTATE API can only check state conditions. See also *Guard expressions*, page 166.

In the various Validator windows you can see what has happened to actions, states, events, variables, etc., as a consequence of the sending of the event. See *Viewing elements during simulation*, page 164.

ACTIVE EVENTS

An active event is an event, which, if sent, will trigger one or more transitions.

Active events are shown in the Event window with red arrowheads (see *Figure 134*, page 163). To view only the active events, choose *Only Active Events* on the Event window pop-up menu.

If the Project contains more than one System, and a global event is active in more than one System, the arrowhead is double. See *Event window*, page 153.

USING THE WATCH WINDOW FOR SENDING EVENTS

- 1 In the Event window, select the event to be added to the Watch window, open the pop-up menu, and choose *Add to Watch*.
- 2 In the Validator or Target view of the Watch window, select the event to send and press ENTER.

Viewing elements during simulation

When an event has been sent, a number of visualSTATE elements will be affected. Via the Validator windows, you can see changes in the following elements:

- States
- Actions
- Assignments
- Signals
- Guard expressions
- Declared elements.

STATES

The states that became current upon sending an event, and the states that were current before the event was sent can be viewed in the System window (for example, use the *Only Current* and *Show Previous* commands of the System window pop-up menu. See *System window*, page 152).

States that became current upon sending the event are shown with a red arrow.

It is possible to filter the information in the System window using the pop-up menu as follows:

- To hide all states that are not current, choose ***Only Current***.
- To see if a state was current before the last deduction was performed, choose ***Show Previous***.

Note: The ***Show Previous*** command is not available when the window is shown in target mode.

- To view a branch, select any state, open the pop-up menu, and choose ***New Branch***. In this way, it is possible to watch only the part of the System that is interesting at the moment.

Or choose ***Add*** from the pop-up menu to add an entire System/instance as a new branch.

- To hide a branch, select a branch topstate, open the pop-up menu and choose ***Hide Branch***.

Or select the branch topstate, and press the DELETE key.

States can also be viewed in the Watch window (see *Watch window*, page 157).

ACTIONS

Actions, or outputs, produced by the sent event are listed in the in the Action window which also lists the arguments with which the actions were called.

The order in which the outputs are listed is runtime specific, meaning that the top-most output was the first output given. This applies to Systems too if the Project contains more than one System. What actually happens is that every time a deduction (microstep) is started for a specific System/instance, the Action window is cleared for outputs coming from that System/instance, and every time an output is given during deduction (microstep), the output is added to the end of the list. For information on microsteps, see *Microsteps and macrosteps*, page 224.

ASSIGNMENTS

Assignments performed due to the sending of an event are shown in the Action window.

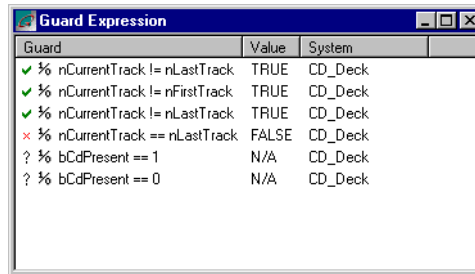
SIGNALS

If the System is using signals, the Signal Queue window will show the new signal queue for the System. See also *Signal queue handling*, page 167

GUARD EXPRESSIONS

To view guard expressions, open the Guard Expressions window.

Note: The Guard Expression window cannot be changed to target mode.



Guard	Value	System
✓ % nCurrentTrack != nLastTrack	TRUE	CD_Deck
✓ % nCurrentTrack != nFirstTrack	TRUE	CD_Deck
✓ % nCurrentTrack != nLastTrack	TRUE	CD_Deck
✗ % nCurrentTrack == nLastTrack	FALSE	CD_Deck
? % bCdPresent == 1	N/A	CD_Deck
? % bCdPresent == 0	N/A	CD_Deck

Figure 135: Guard Expression window (Validator)

In the target application, a guard expression is evaluated during deduction. Consequently it can only have the value TRUE or FALSE.

However, the Guard Expression window of the Validator provides a view of the guard expression values between deductions. This means that a guard expression can also have the value N/A (not available). It will have this value if any unresolved variables, action functions or event parameters are included in the guard expression. If an unresolved guard expression is met during a deduction in the Validator, a dialog box will be displayed where you can specify the value of the unresolved variable.

DECLARED ELEMENTS

To view all variables, all action functions, and all constants declared in all Systems, you use the Variable window. Via the pop-up menu of the window, you can:

- Show or hide a specific group of elements (internal and external variables, actions or constants).
- Show or hide all variables shown in the window, according to location of their declaration (System or Project).
- Hide a range of selected variables. Choose *Hide*.
- Make all hidden variables visible again. Choose *Show All*.
- If a variable is declared as an array, you can display all indexes in the array by choosing *Expand*.

See Figure 144, page 176.

You can hide all other columns than the name column in the Variable window by clicking the appropriate item in the Field Chooser window (see *Field Chooser window*, page 156).

Specifying event parameters

For specifying event parameters, you use the Event window.

- 1 In the Event window activate the pop-up menu and choose *Set Parameter Values....* A dialog box will be displayed. See *Figure 136*, page 167.

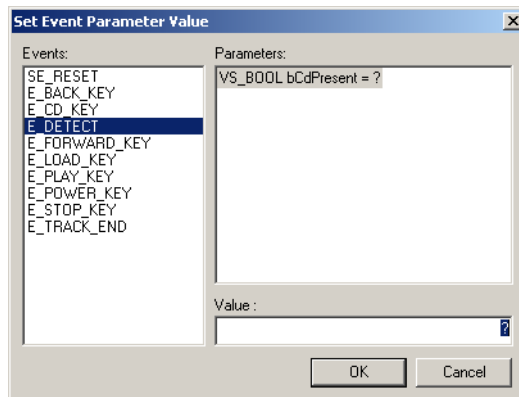


Figure 136: Set Event Parameter Value dialog box (Validator)

- 2 Specify the event parameters, either in the value field of the dialog box, or by editing the parameter label directly.

Note: Event parameters for a target event (target mode) can only be modified from within the Watch window (see *Watch window*, page 157).

Signal queue handling

Every time a signal is sent during a deduction, the signal is added to the end of the appropriate signal queue. Thus the first signal listed in the queue is the one to be sent next. The last signal listed in the queue is the signal added last to the queue (FIFO, first in, first out).

Note: If the signal queue for a specific System/instance is not empty, it is not possible to send an event to that System/instance.

AUTOMATIC VS. MANUAL SIGNAL QUEUE HANDLING

Handling of signal queue can be automatic or manual. See *Activating automatic signal queue handling*, page 168, and *Using manual emptying of signal queue*, page 168.

Note (applies to the model when in Validator mode): If the Project contains more than one System, there is a significant difference between the two approaches to emptying the queue:

- When using *automatic signal queue handling*, the queue of a specific System will be emptied just after the deduction of a Send Event action has been made, and actually before the event is sent to any other enabled Systems.
- When using *manual emptying of signal queue*, the queue is not emptied until event deduction has been performed for all enabled Systems.

If assignments are used, choice of approach may give quite different results.

Activating automatic signal queue handling

You activate automatic signal queue handling as follows:

- 1 Open the Signal Queue window.
- 2 Open the pop-up menu and choose Debug>Auto Empty Signal Queues.

After a deduction the Validator will send the first signal in the queue. As long as there are signals in the queue for the particular System, deduction will continue, and new signals may then be added to the signal queue. If the System is designed with many signals, this process may take a while.

When automatic signal queue handling is applied, microsteps are not available in target mode.

Note: The System may be in a livelock meaning that the signal queue will never be emptied. If a livelock occurs, press ESCAPE to stop sending signals. In target mode, a livelock cannot be stopped.

Using manual emptying of signal queue

You can manually empty signal queues as follows:

- Continue to send the top signal in the queue until the queue is empty. This is done by double-clicking the signal in the Signal Queue window.

Or

- Single-step the queue by choosing *Send Signal* on the Signal Queue Window pop-up menu. This will send the top signal in the first queue containing signals. The order in which the queues are emptied is defined via the System setup (see *System setup*, page 178).

HANDLING SIGNAL QUEUES FOR A SINGLE SYSTEM

You can handle signal queues for a single System as follows:

- 1 Open the Signal Queue window.
- 2 Empty the signal queue:

To empty the signal queue for a specific System, select the System, open the pop-up menu and choose *Empty System Signal Queue*.

To step the signal queue for a specific System, select the System, open the pop-up menu and choose *Send System Signal*.

Breakpoints

It is possible to set up breakpoint conditions for one or more of the following:

- The sent event or signal.
- An expression. It is possible to specify an expression to be evaluated before a deduction, and/or an expression to be evaluated after a deduction.
- The current state (the state before the deduction).
- The next state (the state after the deduction).
- The actions executed during a deduction.

Note: Breakpoints are not available in target mode.

DEFINING BREAKPOINTS

- 1 On the Validator menu, choose Edit>Breakpoints. A Breakpoints Setup dialog box will be displayed. See *Figure 137*, page 170.

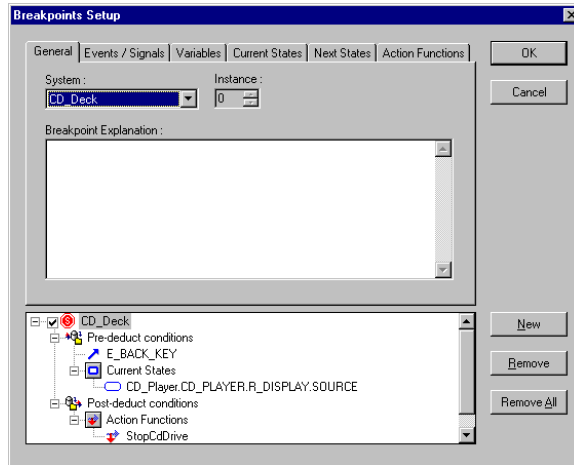


Figure 137: Breakpoints Setup dialog box, General tab (Validator)

- 2 Click the General tab and select the System and instance on which the break should be performed (in the Validator, a breakpoint works on a System).
- 3 Enter an explanation for the breakpoint (not mandatory).

The bottom section of the Breakpoints Setup dialog box contains an overview of all defined breakpoints. You enable and disable the breakpoints by clicking the check box to the left of the System name.

TIP: Enabling and disabling of breakpoints can also be done in the Breakpoints window (opened via View>Breakpoints). The Breakpoints window also contains an overview of the defined breakpoints.

- 4 Create a breakpoint by clicking the **New** button. A new breakpoint is added to the list.
A break can be made on one or several conditions. If more than one condition is defined for a breakpoint, they all have to be fulfilled before a break is performed.
- 5 Set up breakpoint conditions by clicking the appropriate condition type (events/signals, variables, etc.), and subsequently clicking the item to apply as condition. For detailed information, see

- *Assigning a signal or an event as a condition to a breakpoint*, page 171
- *Assigning an expression to a breakpoint*, page 172

- *Setting up breakpoints for specific states*, page 173
 - *Setting up breakpoints for executed actions*, page 174.
- 6 To remove breakpoints, click the **Remove** button or **Remove All** button.

Assigning a signal or an event as a condition to a breakpoint

You assign a signal or an event as a condition to a breakpoint as follows:

- 1 Choose Edit>Breakpoints. In the Breakpoints Setup dialog box, click the Events/Signals tab. See example in *Figure 138*, page 171.

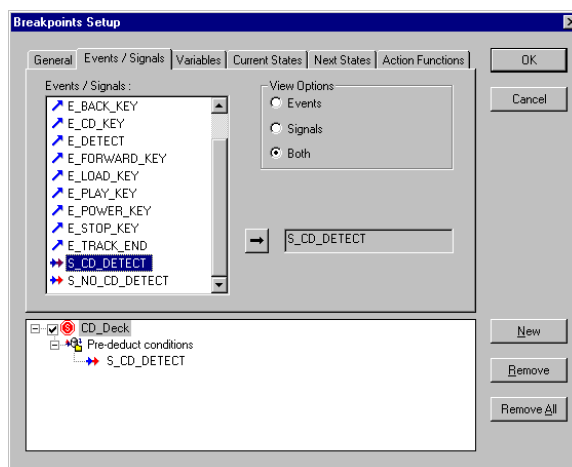


Figure 138: Breakpoints Setup dialog box, Events / Signals tab (Validator)

- 2 Select the appropriate View options (**Events**, **Signals**, or **Both**).
- 3 In the list, double-click the event or signal to assign, or click the Arrow button. The item will be moved to the selected field.

Assigning an expression to a breakpoint

You assign an expression to a breakpoint as follows:

- 1 Choose Edit>Breakpoints. In the Breakpoints Setup dialog box, click the Variables tab. See *Figure 139*, page 172.

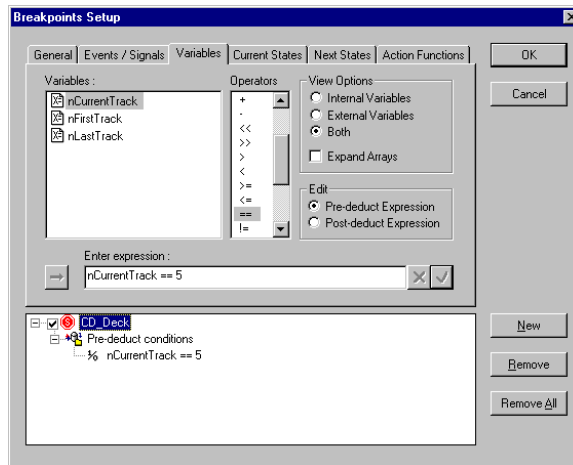


Figure 139: Breakpoints Setup dialog box, Variables tab (Validator)

- 2 Enter a guard expression and click the check mark button. The expression entered must follow the same syntax as that of guard expressions defined in the Designer. It is possible to have an expression evaluated both before and after a deduction is performed. Both expressions are defined within this tab.

Note: It is not possible to use action functions or constants in a breakpoint expression for variables.

Setting up breakpoints for specific states

You set up breakpoints for specific states as follows:

- 1 Choose Edit>Breakpoints. In the Breakpoints Setup dialog box, click the Current States or Next States tabs. See *Figure 140*, page 173.

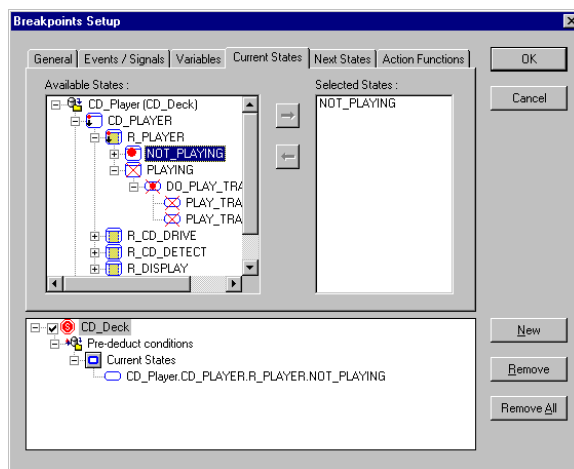


Figure 140: Breakpoints Setup dialog box, Current States tab (Validator)

- 2 Double-click the states for which breakpoint conditions should be applied.

Note: The states defined in the *Current State* tab will be evaluated before a deduction is performed, and the states defined in the *Next State* tab will be evaluated after the deduction.

Setting up breakpoints for executed actions

You set up breakpoints for executed actions as follows:

- 1 Choose Edit>Breakpoints. In the Breakpoints Setup dialog box, click the Action Functions tab. See *Figure 141*, page 174.

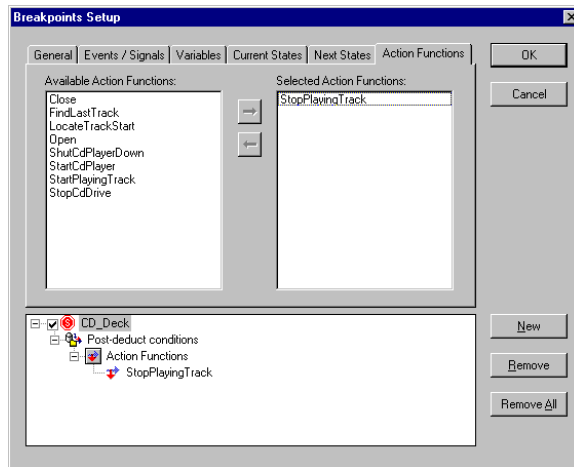


Figure 141: Breakpoints Setup dialog box, Action Functions tab (Validator)

- 2 In the Available Action Functions list, double-click the action functions for which to set up breakpoints.

The order in the lists of selected action functions and the action functions actually executed may differ, but this has no influence on the evaluation of the breakpoint.

USING BREAKPOINTS

The breakpoint pre-deduct conditions are evaluated just before deduction starts. If all conditions are fulfilled, and the breakpoint does not contain any post-deduct conditions, the Breakpoint Reached dialog box will be displayed where you can select one of the following options (see *Figure 142*, page 175):

- Click **Step Over** to step over the breakpoint and thereby perform the deduction.

- Click **Stop**.

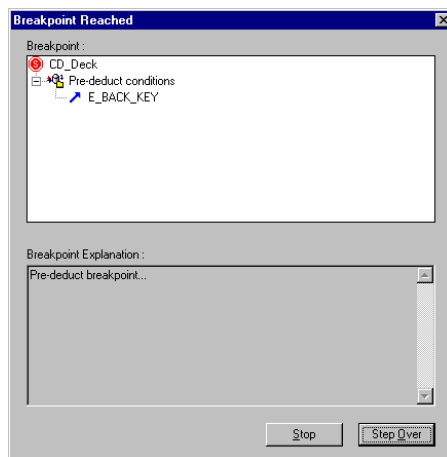


Figure 142: Breakpoint Reached dialog box, Pre-deduct (Validator)

After deduction, all post-deduct conditions are evaluated. If all post-deduct conditions in a breakpoint are fulfilled (and of course all pre-deduct conditions too, but they have been evaluated), a break is performed. Again the Breakpoint Reached dialog box pops up informing you that a breakpoint is reached. See *Figure 143*, page 175.

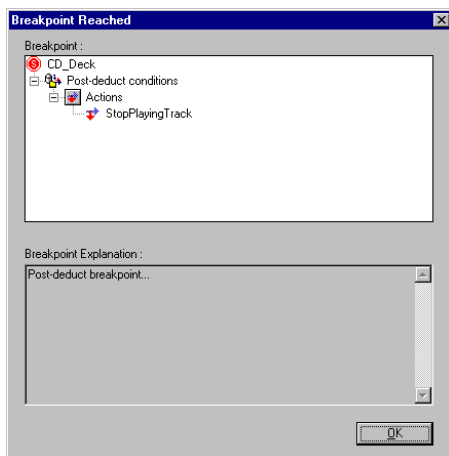


Figure 143: Breakpoint Reached dialog box, Post-deduct (Validator)

For post-deduct conditions, you cannot stop or step over the breakpoint. Click **OK** to exit the dialog box.

Note: If the Project contains more than one System/instance, and you choose to stop on a breakpoint, all further processing is disabled, but nothing is undone.

This means that if

- the Project contains two Systems, and
- a deduction has been performed on the first System, and
- a pre-deduct breakpoint is reached on the second System, and
- you choose to stop

the deduction on the first System will remain.

Changing variable values

You change variable values as follows:

- 1 Open the Variable window.

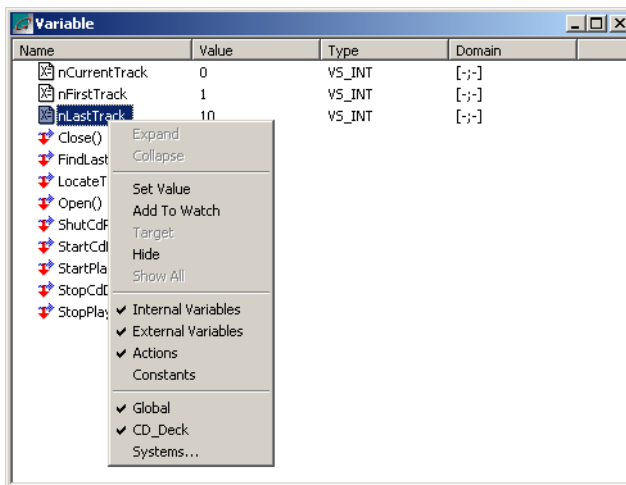


Figure 144: Variable window (Validator), with pop-up menu

- 2 Click the variable for which to change value.
- 3 Open the pop-up menu and choose **Set Value**. Type the new value.

Note: At load time the variables are assigned their initialization values.

Note: Arrays must be expanded before it is possible to set the value of the different indexes.

Setting action function return values

An action function can have a return value. In order to be able to simulate the System, the action function return value may be necessary if the action function return value is used in a guard expression or an assignment expression.

To set the value action function return value:

- 1 Open the Variable window. See *Figure 144*, page 176.
- 2 In the Value column, select the action function return value, and type the value.

Each time an action function is used you can be prompted to specify return value. This is done by choosing **Debug>Action Function Return Value Prompt**. By default the value is undefined.

Note: In target mode, it is not possible to view the action function return values.

Forcing states

It is possible to force the System into a specific state. All states except regions can be forced. You force a state as follows:

- 1 Open the System window.

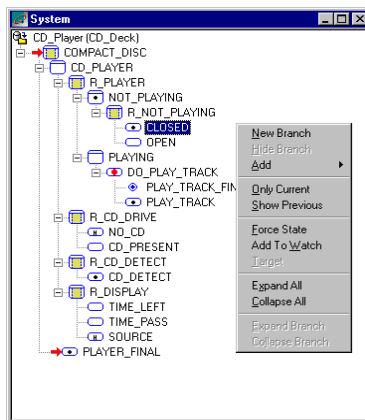


Figure 145: System window (Validator), with pop-up menu

- 2 Click the state to force. Open the pop-up menu, and choose **Force State**.

The state will become active in the System.

System setup

Because the Validator is able to handle simulation of more than one System and even of Systems that contain multiple instances, it is possible to set up the order of Systems, as follows:

- 1 Open the System Setup window. See *Figure 146*, page 178.

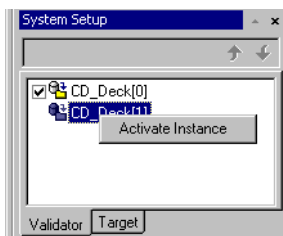


Figure 146: System Setup window (Validator)

- 2 Click the Validator or Target tab to change between Validator and RealLink setup.
- 3 Change System order by clicking the Up Arrow or Down Arrow buttons on the toolbar.

Changing the order of the System setup changes the order of how events are sent to the different Systems. Thus it is possible to match the handling of events by the target application as closely as possible. Furthermore changing the System order affects the handling of signal queues. If manual signal queue handling is used (see *Using manual emptying of signal queue*, page 168), System setup determines which queue should be emptied first.

Note: The System order only applies to interactive simulation (simulation not using test sequence files). When a recorded test sequence is played, all inputs to the Systems are performed on a System/instance basis, and it makes no sense to manually empty a signal queue. See *Recording and playing test sequences*, page 187.

- 4 Enable or disable Systems by clicking the check boxes to the left of the System name. Disabled Systems will not receive events.

To activate an instance (only possible in Validator mode):

- 1 In the System Setup window, select the instance to be activated.
- 2 Open the pop-up menu, and choose *Activate Instance*.

See *Figure 146*, page 178.

Note: It is not possible to change instances in target from within the Validator.

Graphical animation

It is possible to view a Validator simulation graphically in the Designer. When the Designer is used in this way, it is said to be in *simulation mode*, and the System design cannot be changed.

To view a simulation graphically:

- 1 On the Validator menu, choose Debug>Graphical Animation, or click the Graphical Animation button on the Debug toolbar. This launches the Designer in simulation mode. See *Figure 147*, page 179.

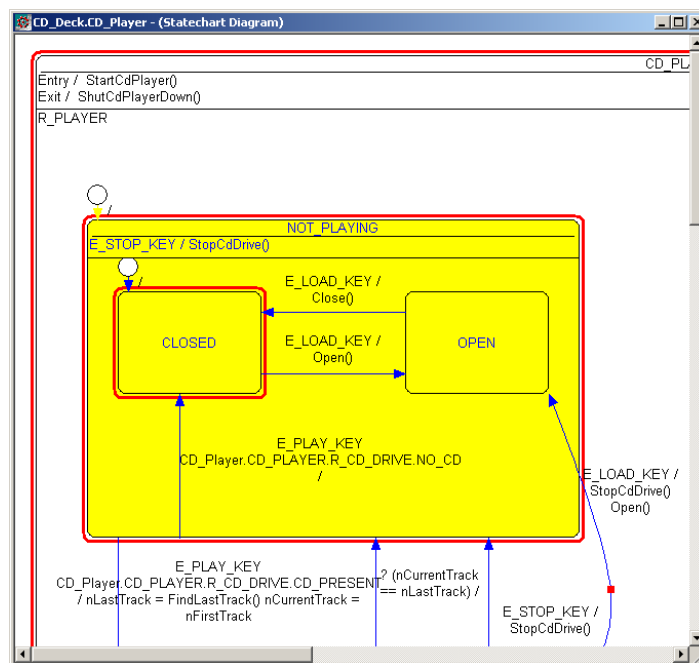


Figure 147: Example of graphical animation

- 2 In the Designer, open the diagram to view.

When a transition fires in the Validator, the affected states and transitions can be viewed in the Designer. All opened diagrams are updated each time a microstep is completed. For information about microsteps, see *Microsteps and macrosteps*, page 224.

SETTING BREAKPOINTS FOR GRAPHICAL ANIMATION

You can set breakpoints for graphical animation as follows:

- 1 From the Validator, open the Designer in simulation mode (see *Graphical animation*, page 179).
- 2 Select the state for which to apply breakpoints, and open the pop-up menu.
- 3 Select *Insert/remove current state breakpoint*, or *Insert/remove next state breakpoint*, whichever is appropriate.

To delete all breakpoints, select *Remove all breakpoints*.

SETTING GRAPHICAL ANIMATION OPTIONS

You can customize the graphical settings for the elements displayed in the Designer simulation diagrams as follows:

- 1 Open your Validator workspace. On the Validator menu, choose Debug>Graphical Animation. The Designer will be launched in simulation mode.
- 2 On the Designer menu, choose Tools>Configure....
- 3 A Configure Simulation dialog box will be displayed. Set shape and color of bounding frames, and specify whether previous current states should be shown in the simulation diagram.

Toggling between Validator mode and target mode

When the Validator is connected to a target by means of RealLink (target mode), you can change the mode of the windows so as to view the representation of the Validator model or the runtime model. All windows have this option, except the Guard Expression window. For a detailed description of RealLink, see *Testing visualSTATE models using RealLink*, page 207.

To change a Validator window so as to view the runtime model (RealLink):

- 1 Open a window, for example the Event window.

- Click in the window and choose **Target** from the pop-up menu (see *Figure 148*, page 181), or press ALT+F8.

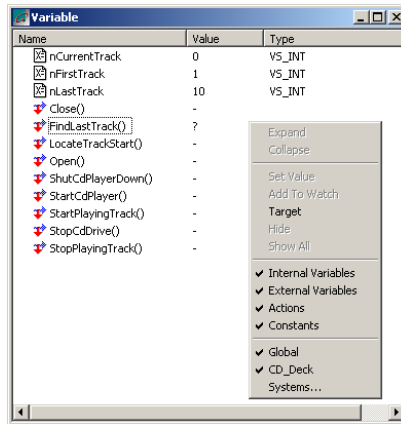


Figure 148: Target command in Validator window

Tracing visualSTATE models

This chapter shows how to trace visualSTATE models in the Validator.

Tracing

A trace is a sequence of steps that leads to a desired configuration. Tracing can be used for answering the question “How do I get from the initial state to a user defined configuration?”.

The Validator can be used for setting up the configuration you want to reach, and you can see the resulting trace by using the Validators capability for handling test sequence files. Refer to “Playing recorded test sequences” on page 191 for how to use the resulting test sequence.

The Verificator will be used for finding the actual trace. In a trace the Verificator will find a suitable sequence of events and external variable values that make it possible to reach the desired configuration.

SETTING UP A TRACE

- 1 Launch the Validator and open your Validator workspace.
- 2 On the menu, choose Debug>Trace to open the Trace Setup window.
- 3 There are 3 options for setting the desired configuration to reach. See *Figure 149*, page 183.

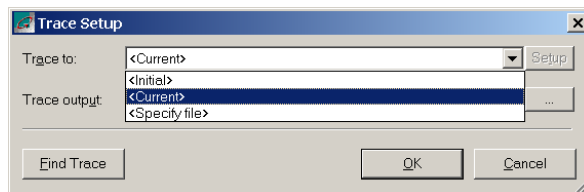


Figure 149: Trace Setup, Trace To options

You may set the Trace To to:

- Initial - this will make a trace to the initial state in the System.
- Current - this will make a trace to the current state in the System.

- **Specify file** - this is used if you want to specify a customized setup

Selecting 'Initial' or 'Current' completes step 3, but if you select 'Specify file' you need to setup the desired configuration of states. Simply click **Setup** like shown in *Figure 150*. Then refer to "Setting up the Trace Point" on page 184 for how to handle Trace Point Setups.

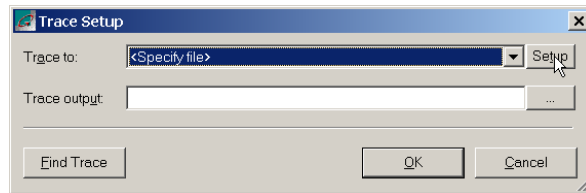


Figure 150: Trace Setup, Trace To Setup

- 4 After completing the 'Trace To' part you need to select which file the resulting test sequence file should be saved to. Write the desired output file name in the 'Trace output' field or browse for the file using ... to the right of the field.
- 5 After completing the previous steps select **Find Trace**. The Validator will by means of the Verificator find a trace to the specified state configuration and the resulting test sequence file will be saved, if a trace can be found.

SETTING UP THE TRACE POINT

A trace point is the state configuration you want to reach. You can open an existing Trace Point Setup, save one and configure one. This is done in the 'Trace Point Setup' window which is opened as shown in "Setting up a trace" on page 183.

The trace point setup window looks like *Figure 151*, page 185.

The window has 3 buttons for quick handling of the configuration:

- **Initial** - sets the state configuration to the initial state(s) in the System.
- **Current** - sets the state configuration to the current state(s) in the System.
- **Clear** - clears the state configuration.

The **Load**, **Save** and **Save As** are used in the normal way in Windows for loading, saving, and saving under another or a new name for the trace point setup files.

If your Project contains more than one System you will also be able to select which System you want to trace by means of the 'Select System' drop-down list.

Select your desired trace point by selecting states in the window. When done save the trace point you have set up and click **OK**.

The saved trace point file will be saved with information on the System as well, so you can use this information when you want to retry a trace later on. If you change the System you will not be able to reuse the trace point since the signature of the System will be checked. Likewise you will not be able to use a trace point file made for another System for the current System in the Validator.

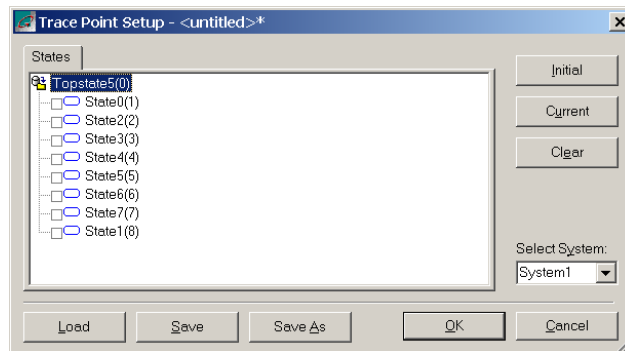


Figure 151: Trace Point Setup

Recording and playing test sequences

This chapter describes how you can record test sequences to test sequence files by means of the Validator, and how you can play and change the recorded test sequences.

Recording a test sequence

It is possible to record one or more test sequences to a test sequence file. The test sequence file can be used as a source of reference in future simulation sessions, for example after a change in model design.

A test sequence consists of a number of steps. Each step describes the command given, to where it is given (if applicable), and the output produced by the command.

To record a test sequence to a test sequence file:

- 1 Launch the Validator and open your Validator workspace.
- 2 Choose File>Test Sequence File>New. A Test Sequence File window is displayed.

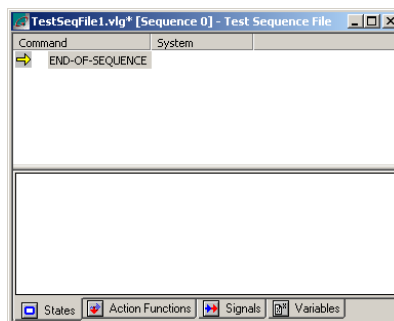


Figure 152: Validator Test Sequence File window

- 3 Click the Record button on the Debug toolbar, or choose Debug>Record on the menu.
- 4 Initialize the loaded Systems, and send the visualSTATE reset event `SE_RESET`. This will ensure that the starting point is always the same when test sequences are played. If you do not start by initializing the visualSTATE Systems, you will get an error.

- 5 Apply commands to the System. The commands that can be given to a System and recorded in a test sequence file are listed in *Table 5*, page 188.

Command	Do the following
Initialize a System	Click the Initialize button on the Debug toolbar (not available in target mode).
Send an event into the System.	Double-click an event in the Event window.
Set the values of internal and external variables, and action return values.	Open the Variable window pop-up menu and choose Set Value (values of action return values are not available in target mode). See also <i>Changing variable values</i> , page 176.
Force the System into a specific state.	Open the System window pop-up menu and choose Force State (not available in target mode).
Send a signal into the System.	In the Signal Queue window, double-click a signal.

Table 5: Commands that can be recorded to a Validator test sequence file

The commands applied to the visualSTATE model will be recorded and appended to the selected sequence (for selection of test sequence see *Playing recorded test sequences*, page 191).

If manual (interactive) simulation is performed on multiple Systems, global events are sent to all Systems and will be recorded once for each System receiving the event. This way of recording ensures that it is possible to repeat the test sequence by playing it (see *Playing recorded test sequences*, page 191).

Note: If you are recording a test sequence, *all* commands applied to the model will be recorded, both manually applied commands and commands applied automatically by means of a recorded test sequence file.

- 6 Stop recording by clicking the Record button on the toolbar.

VIEWING OUTPUTS OF STEPS

The outputs of steps (commands) recorded to a test sequence file during a simulation session can be viewed by selecting the appropriate command (step) in the Test Sequence File window. The recorded (expected) output of the selected command will be displayed in the lower part (output section) of the window. See *Figure 153*, page 189.

If the output section of the Test Sequence File window is not visible, open the pop-up menu of the window and choose **Step results**. Click the tabs to change between the output types (see *Output types*, page 189).

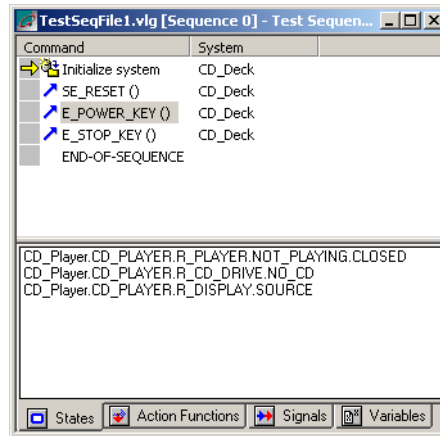


Figure 153: Validator Test Sequence File window, output of selected command

The upper part of the window shows the commands (steps) recorded and the System to which they were applied. The test sequence pointer (arrow in left-most column) shows the step reached in the sequence being played.

Output types

Outputs are divided into the following types (see *Figure 153*, page 189):

States	The entire state configuration for the System to which the command was given.
Action functions	The action functions executed during a <i>Send Signal</i> or a <i>Send Event</i> command.
Signals	The entire queue after a <i>Send Signal</i> or a <i>Send Event</i> command.
Variables	The variables that have been assigned a new value during a <i>Send Signal</i> or a <i>Send Event</i> command (not necessarily another value, but an assignment has been performed to the variable).

Note: Not all commands produce all four output types.

See also *Comparing played test sequences with recorded output*, page 193.

COLLECTING TEST SEQUENCES IN THE SAME FILE

It is possible to collect different test sequences in the same test sequence file, and give each sequence a specific name. For handling test sequences, you use the pop-up menu of the Test Sequence File window (see *Figure 154*, page 190).

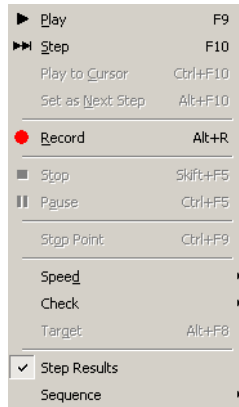


Figure 154: Pop-up menu of Validator Test Sequence File window

Creating and deleting test sequences in a file

To create a new sequence, choose *Sequence>New Sequence*. Click the Record button and apply commands to the System. Click the Record button to stop recording. The existing sequence is saved automatically.

To delete all steps in the current sequence, choose *Sequence>Reset Sequence*.

To enter a name and explanation for the test sequence, choose *Sequence>Select Sequence*. Type name and explanation in the dialog box displayed. See *Figure 155*, page 190.

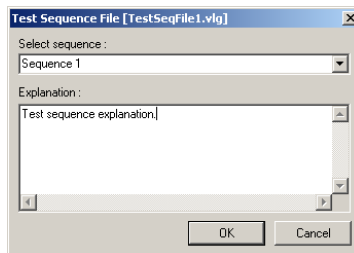


Figure 155: Test Sequence File dialog box (Validator)

To record a test sequence from the target, select **Target**. Click the Record button and apply commands to the System. Click the Record button to stop recording.

To delete the current sequence, choose **Sequence>Delete Sequence**.

Note: At least one sequence must exist in the file so it is not possible to delete *all* sequences in the file.

Opening test sequences

To open the next or previous sequence, choose **Sequence>Next Sequence** or **Sequence>Previous Sequence**.

To open a sequence that is not previous or next, choose **Sequence>Select Sequence**. This will open a dialog box where you can select the sequence by clicking in the list box. See *Figure 155*, page 190.

Playing recorded test sequences

It is possible to play recorded test sequences from test sequence files in the Validator. This allows you to check if two different simulation sessions give the same result, for example after a design change. Once an appropriate set of test sequences has been created, they can be used repeatedly to check that design changes result in expected behavior of the model. The test can also be repeated for the target model using RealLink.

To play a recorded test sequence:

- 1 Open your Validator workspace.
- 2 Choose **File>Test Sequence File> Open**, and specify the file to use. The test sequence file window is displayed. Select the test sequence to apply. See *Figure 155*, page 190. On the pop-up menu, choose **Step results** to open the output section of the window, if it is not already open.
- 3 Set the starting step in the sequence: Select the step in the upper part of the Test Sequence File window, and choose **Set as Next Step** from the pop-up menu. Or click the **Stop** button on the Debug toolbar.
- 4 To play the recorded test sequence automatically, click the Play button on the Debug toolbar.

To execute the steps in the test sequence one by one, click the Step button on the Debug toolbar.

SPEED

If a test sequence is executed automatically, speed can be set. Default speed is Free Run which is the highest possible speed of the PC.

To set a different speed, choose Edit>Speed, or use the pop-up menu of the Test Sequence File window.

BREAKING EXECUTION OF A TEST SEQUENCE

It is possible to break the execution of a test sequence in a test sequence file, as follows:

- 1 Open the test sequence file and select the sequence.
- 2 If you know exactly on which step to break execution, either select the step and subsequently choose *Play to Cursor* from the Test Sequence File pop-up menu. Or set a stop point on the specific step by double-clicking it.

To search for some specific conditions to be fulfilled, use breakpoints (see *Breakpoints*, page 169). Breakpoints also work for commands sent from a recorded test sequence.

To pause execution, click the Pause button on the Debug toolbar.

To stop execution and return the cursor to the first step in the sequence, click the Stop/Reset button on the Debug toolbar.

JUMPING TO A SPECIFIC STEP IN A RECORDED TEST SEQUENCE

It is possible to jump around in a recorded test sequence via the *Set as Next Step* command on the pop-up menu of the Test Sequence File window (see *Figure 154*, page 190).

This is particularly useful if the signal queue in a recorded test sequence does not correspond to the one generated at runtime. Execution of the recorded test sequence will stop if the sequence tries to send a signal different from the first signal in the queue. To continue execution in such a situation, do the following:

- 1 Open the test sequence file and select the sequence.
- 2 As the next command to be executed, select the first command not being a signal in the Test Sequence File window
- 3 Manually empty the existing queue by clicking the Empty Signal Queues button on the Debug toolbar (or use the *Empty Signal Queue* command on the Test Sequence File window pop-up menu).

It will now be possible to continue playing the test sequence file.

COMPARING PLAYED TEST SEQUENCES WITH RECORDED OUTPUT

It is possible to have the output (states, action functions, signals, and variables) of a played test sequence compared with the recorded output, as follows:

- 1 Open the Test Sequence File window, and choose **Check** from the pop-up menu. Select the items for which to check. By default all four output types are selected (for a description of the output types, see *Output types*, page 189).
- 2 Play the sequence, as described in *Playing recorded test sequences*, page 191.

If a design change has been made that results in mismatches, and you play the recorded test sequence, execution will stop, and the Validator will report the mismatches caused by the change. See *Figure 156*, page 193.

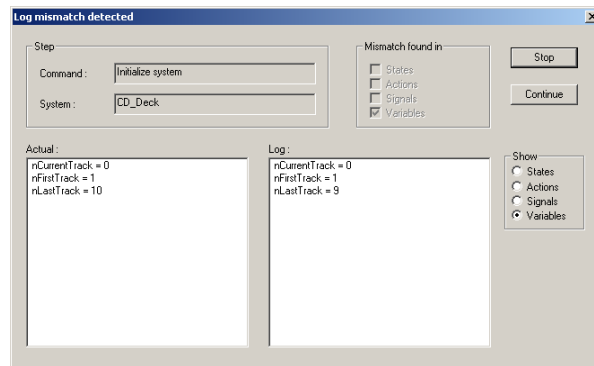


Figure 156: Log Mismatch Detected dialog box (Validator)

Analyzing visualSTATE models

This chapter how to analyze visualSTATE models with regard to elements used and test coverage. The types of analysis are termed *static analysis* and *dynamic analysis* respectively.

Static analysis

A static analysis gives an overview of the elements used in the transitions of a specific state machine model. For example an answer to the question “Which transitions will fire the action *a*?” or “Which transitions involve the variable *v*?”.

The static analysis information can be obtained without executing or simulating the state machine model.

The elements for which transitions can be statically analyzed are:

- Events
- Actions
- Signals
- Internal Variables
- External Variables.

PERFORMING A STATIC ANALYSIS

- 1 Launch the Validator and open your Validator workspace.
- 2 On the menu, choose File>Analysis>New Static to open a Static Analysis window.
- 3 On the Analysis toolbar, select the System on which to perform the analysis. See *Figure 157*, page 195 where the selected System is *CD_Deck*.



Figure 157: Validator Analysis toolbar, static analysis

- 4 In the left pane of the Static Analysis window, select the elements for which to analyze transitions (hold the CTRL button down while clicking the left mouse button on the elements).

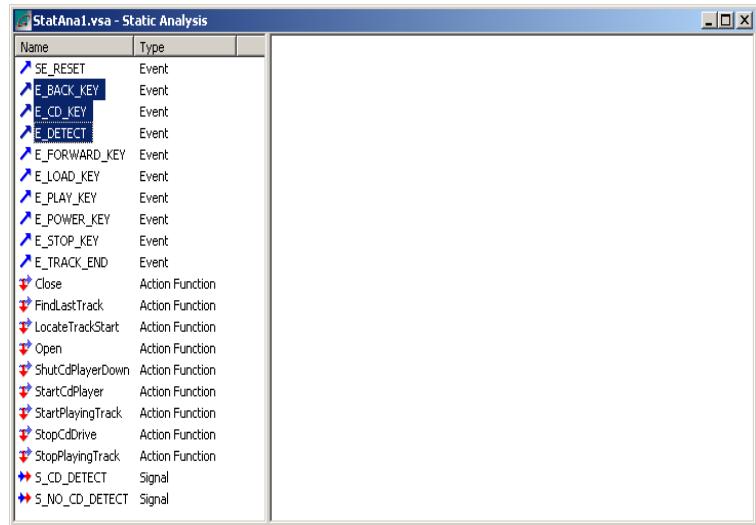


Figure 158: Validator Static Analysis window, selection of elements to analyze

- 5 On the Analysis toolbar, click the Analyze button, or choose Debug>Analyze.

Analysis will be performed and analysis results shown in the right pane of the Static Analysis window.

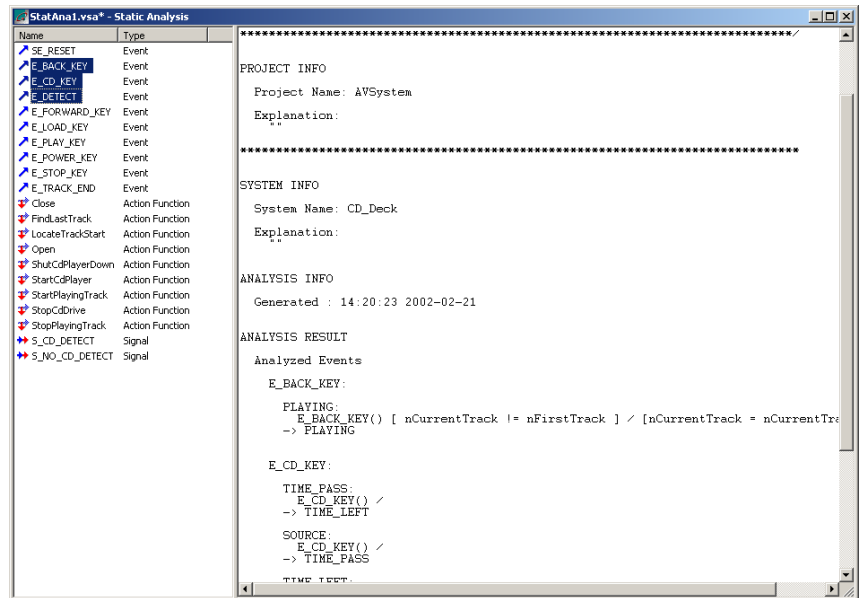


Figure 159: Static analysis results (Validator)

- 6 Save the static analysis file by choosing File>Analysis>Save. Specify file name and directory in the Save Static Analysis dialog box displayed.

OPENING AN EXISTING STATIC ANALYSIS FILE

You can open an existing static analysis file by choosing File>Analysis>Open. In Open Analysis File dialog box displayed, specify file name (extension `vsa`) and directory.

Dynamic analysis

A dynamic analysis calculates the test coverage of a specific System and includes events, actions, signals, conditional states, next states and transitions. The test coverage analysis gives detailed information on the dynamic aspects of the model when specific scenarios or parts of the model are simulated.

For example a dynamic analysis will describe which parts of the model have the highest activity level, and which parts are never entered. This information is useful when analyzing how the dynamics of the application will perform at runtime.

PERFORMING A DYNAMIC ANALYSIS

- 1 Launch the Validator, open your Project in a Validator workspace, initialize the System, and send events into the System by double-clicking them (see *Simulation*, page 161).
- 2 On the Validator menu, choose File>Analysis>New Dynamic. An empty Dynamic Analysis window is displayed.
- 3 On the Analysis toolbar, select the System on which to perform the analysis. See *Figure 160*, page 198 where the selected System is CD_Deck.

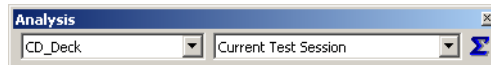


Figure 160: Validator Analysis toolbar (dynamic analysis)

- 4 On the Analysis toolbar, select the sequence for which to perform the analysis. This can be a sequence in a test sequence file, or it can be performed on the data collected since the last time the dynamic analysis data was reset. This set of data is named *Current Test Session* (see *Figure 159*, page 197). Using collected data allows an on-the-fly calculation of the test coverage.
- 5 On the Analysis toolbar, click the Analyze button.

Analysis will be performed and analysis results shown in the Dynamic Analysis window. See *Figure 161*, page 199.

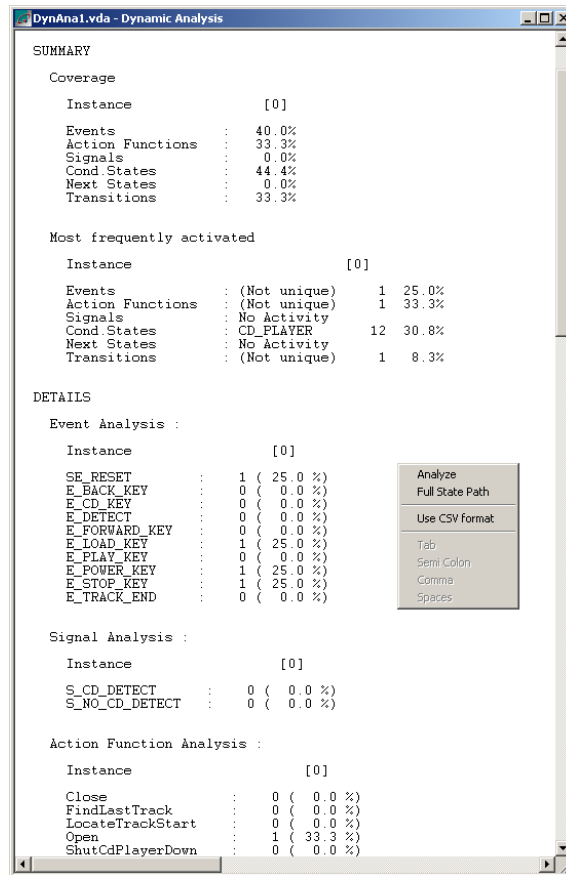


Figure 161: Validator Dynamic Analysis window, with pop-up menu

The dynamic analysis consists of a summary section and a details section. The summary section shows the calculated coverage percentage and the most frequently activated elements of those covered by the analysis. In the details section it can be seen how many times a specific element has been activated. Furthermore the dynamic analysis calculates frequency as a percentage of the entire activation of this group of identifiers.

The result of the dynamic analysis can be in either text format or comma separated values format (CSV). Format is selected via the pop-up menu of the Dynamic Analysis window.

Note: The dynamic analysis data is reset each time a dynamic analysis is performed, and each time Edit>Undo is *applied to a **Send Event** or **Send Signal*** command.

- 6 Save the dynamic analysis file by choosing File>Analysis>Save. Specify file name and directory in the dialog box displayed.

OPENING AN EXISTING DYNAMIC ANALYSIS FILE

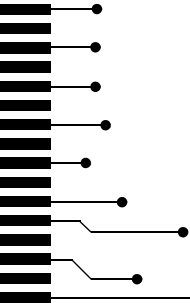
You can open an existing dynamic analysis file by choosing File>Analysis>Open. In the dialog box displayed, browse for directory and specify file name (extension `.vda`).

Part 6: Testing in target applications

This part of the visualSTATE[®] User Guide includes the following chapters:

- Introduction
- Testing visualSTATE models using Reallink.





Introduction

With the RealLink facility of the Validator it is possible to monitor and control the runtime behavior of your visualSTATE model in the target application.

This chapter gives an introduction to visualSTATE RealLink, and describes

- RealLink connection to target
- visualSTATE elements supported by RealLink
- Target requirements.

For a description of how to use RealLink, see *Testing visualSTATE models using RealLink*, page 207.

What is RealLink?

visualSTATE RealLink comprises the software running on the PC, the target and a communication link between the two. See *Figure 162*, page 204.

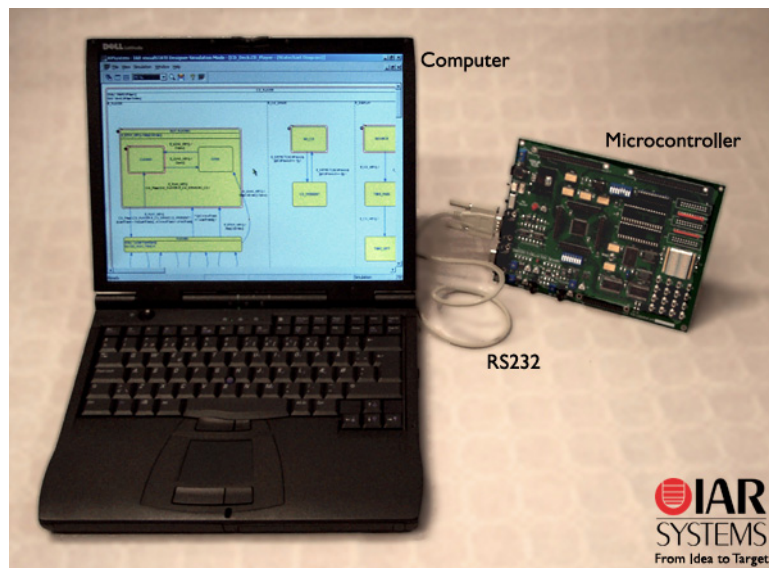


Figure 162: Example of visualSTATE RealLink setup

RealLink connection to target

The connection between the Validator and target is established by means of a communication module as shown in *Figure 163*, page 205. RealLink supports multiple communication modules that each provides an interface to a specific link to the target, such as a serial connection (RS232), Bluetooth™ connection, TCP/IP connection or any other type of communication link.

Each communication module automatically integrates itself into the Validator via a communication plugin (DLL). visualSTATE includes the following communication plugins for RealLink:

- RealLink RS232 communication plugin (see *Setting up RealLink RS232 communication plugin*, page 217).

- RealLink TCP/IP communication plugin (see *Setting up RealLink TCP/IP communication*, page 218).

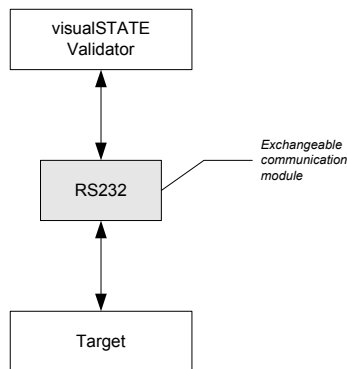


Figure 163: RealLink connection between the Validator and target

An example of a visualSTATE RealLink setup is shown in *Figure 162*, page 204.

Once the RealLink connection is established, you have full control of the visualSTATE model running in the target. From within the Validator, events can be sent to the target, test sequence files can be recorded and played, and variables can be changed, all in real-world hardware.

visualSTATE elements supported by RealLink

With RealLink it is possible to monitor and control the behavior of all logical visualSTATE elements, except the following:

- Parameters to action functions: Their values are shown as "..." in the Validator Action window.
- Guard expressions of active events: The Validator Event window shows the active events but no guard expressions are considered. Therefore the Validator may show an event as being active when in fact a guard expression is not satisfied.
- Instances: It is not possible to change instances from within the Validator.

For a detailed description of visualSTATE elements, see *visualSTATE Reference Guide*.

Target requirements

Target processors to be used with Validator RealLink must comply with the following requirements:

Variable sizes

Must be a multiple of 8 bit, however max. 32 bit.

Memory

Memory used by RealLink must be accessible through byte pointers. Some memory areas in specific microprocessors have only 16 bit access. These memory areas cannot be accessed by visualSTATE.

RealLink requires additional memory in CODE, CONST DATA and DATA. See *Appendix B: RealLink memory consumption*, page 409.

Communication

The receive function must be interrupt-driven (polled communication is not supported), and RealLink must have exclusive access to the communication resource. The settings of the communication resource must match the settings of the communication module installed on the PC (see *Setting up RealLink*, page 207).

Note: To connect to a target with Harvard architecture, your compiler must be capable of using generic pointers, or you must use extended keywords on RealLink symbol tables. See *Targets with Harvard architecture*, page 208.

visualSTATE Expert API requirements

If more than one VS System is loaded in a given task (or in the main loop if no RTOS is used), the following applies:

- Only one `VS_WAIT()` macro per task.
- An `SMP_Deduct()`, `SMP_GetOutput()`, `SMP_NextState()` sequence must be completed before calling `SMP_Deduct()` a second time.
- If you want to use RealLink all systems should be running in the same task.

Testing visualSTATE models using RealLink

This chapter describes how to set up RealLink and monitor and control the runtime behavior of your visualSTATE model in the target application. For RealLink memory consumption, see *Appendix B: RealLink memory consumption*, page 409.

Setting up RealLink

This section describes how you prepare your target application for using RealLink, and configure the RealLink connection.

CHECKLIST

To get RealLink configured and ready for your project, the following steps must be completed:

Step 1: Enabling RealLink support, page 207.

Step 2: Adding RealLink files to your project, page 209.

Step 3: Using the RealLink API, page 210.

Step 4: Implementing target-specific functions, page 213.

Step 5: Completing the target source code, page 215.

Step 6: Configuring the Validator for RealLink, page 215.

For detailed information about code generation and the visualSTATE APIs, see *Part 7: Code generation*, page 231 of this guide, and *visualSTATE API Guide*.

Step 1: Enabling RealLink support

- I Launch the Navigator, and open your workspace file.

- 2 In the workspace browser (workspace view), select the visualSTATE Project for which to enable RealLink support. Open the pop-up menu and choose Project>Options>Code generation. The Coder Options dialog box is displayed. See *Figure 164*, page 208.

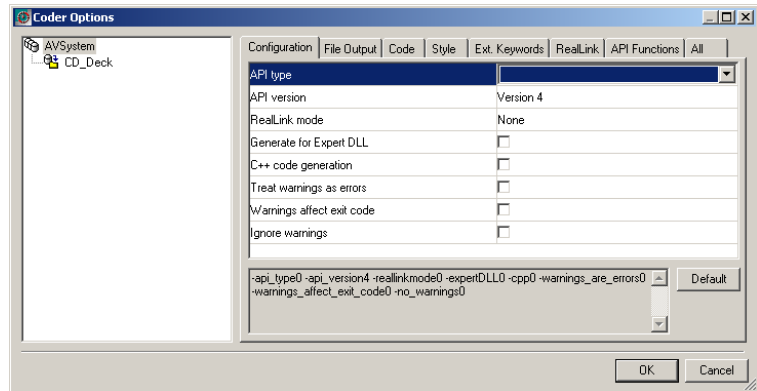


Figure 164: Navigator, Coder Options dialog box, Configuration tab

- 3 On the Configuration tab, select **RealLink mode: Table-based**.
- 4 Click the RealLink tab and set the options appropriate for your Project.
- 5 On the Navigator menu, choose Project>Code generate to generate the source code for the active visualSTATE Project.

Targets with Harvard architecture

To connect to a target with Harvard architecture, your compiler must be capable of using generic pointers, or you can specify extended keywords on RealLink symbol tables as follows:

- 1 Open the Coder Options dialog box of the Navigator, and ensure that **RealLink mode** on the Configuration tab has been set to **Table-based**.

- 2 Click the RealLink tab. See *Figure 165*, page 209.

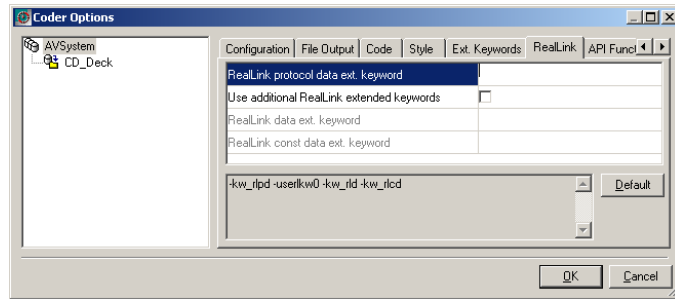


Figure 165: Navigator, Coder Options dialog box, RealLink tab

- 3 Select *Use additional RealLink extended keywords*.
- 4 Click *RealLink data extended keyword* and type a keyword that specifies a memory area where both read and write operations can be performed.
- 5 Click *RealLink const data extended keyword* and type a keyword that specifies a memory area where read operations can be performed.

Note: When you use RealLink extended keywords, the keywords must match the visualSTATE Coder extended keywords. For example, the *RealLink data extended keyword* must match the keywords you specify for external and internal variables in the Coder options.

Step 2: Adding RealLink files to your project

In order to successfully compile and link your project with RealLink support, you must add the following two C modules to your compiler project (or makefile):

- `<SystemName>RealLink.c`
- `<SystemName>VSrlps.c`

The `<SystemName>` prefix is prepended to the filename if the option **Use prefix for API** is being used. The C module `<SystemName>RealLink.c` includes the C header file `<SystemName>RealLink.h`. Include the `<SystemName>RealLink.h` file in the file containing the visualSTATE deduction sequence (a sequence of the visualSTATE API functions `SEM_Deduct`, `SEM_GetOutput`, `SEM_NextState`). See *Examples of main functions*, page 210.

`RealLink.c` and `RealLink.h` are the RealLink API files. These files are generated by the Coder for the Basic API. For the Expert API, the files are static and included with the visualSTATE standard installation. The files are located in the `\IAR Systems\visualSTATE X.x\Api\VSApiRealLink` directory.

Note: This behavior is new from version 5.3. In earlier versions, the RealLink API files were static also for the Basic API.

The `<SystemName>VSRlps.c` file is a Coder-generated RealLink support file. It is placed in the output directory you have specified, together with the other Coder-generated files.

Note: Do not manually edit any RealLink files, because they will be overwritten during the next code generation.

Refer to your compiler manual on how to add additional source files to an existing project.

Step 3: Using the RealLink API

To use the RealLink API, you must make the following changes to your code. (The `<SystemName>` prefix is prepended to the filename if the Basic API is used with the option **Use prefix for API**.)

- 1 Call the Basic API function `SEM_InitAll()`.
(This replaces calls to the Basic API initialization functions such as `SEM_Init()`, `SEM_InitSignalQueue()`, etc.).
- 2 Call the RealLink API function `VS_RealLinkInit()`.
- 3 Insert the RealLink API macro `VS_WAIT(SEM<SystemName>)` in the main loop but before the visualSTATE deduction sequence. The main loop is identified by an infinite loop, typically a `while(1)` or `for(;;)` loop.

Note: The `VS_WAIT()` macro must not be inserted inside the visualSTATE main loop. See *Examples of main functions*, page 210.

When the visualSTATE application enters the `VS_WAIT()` macro, data is exchanged between the Validator and the target. When data exchange is completed, the visualSTATE application program resumes execution, according to your commands from the Validator.

Examples of main functions

Below is an example of a simple visualSTATE Basic API main function and a simple visualSTATE Expert API main function. Each example includes a main loop with a visualSTATE deduction sequence that has been modified to support RealLink. Note that the `VS_WAIT()` macro is inside the main loop, but outside the visualSTATE deduction sequence.

The code that you must insert is shown in **bold**.

Example of Basic API main function

```

/* include Reallink API */
#include "SystemNameReallink.h"

SEM_EVENT_TYPE EventNo;
SEM_ACTION_EXPRESSION_TYPE ActionEx;
unsigned char CC;

int main(void)
{
    /* init Basic API */
    SEM_InitAll();

    /* init Reallink */
    VS_ReallinkInit();

    while(1) /* main loop */
    {
        /* Reallink wait macro */
        VS_WAIT(SEMSystemName);

        /* Get event from queue */
        EventNo = GetEventFromQueue();

        if(EventNo != EVENT_UNDEFINED)
        {
            /* Deduct event */
            if( (CC = SystemNameSEM_Deduct(EventNo)) != SES_OKAY)
                ErrorHandling(CC);

            while( (CC = SystemNameSEM_GetOutput(&ActionEx) == SES_FOUND) )
                SystemNameSEM_Action(ActionEx);
            if(CC != SES_OKAY)
                ErrorHandling(CC);

            if( (CC = SystemNameSEM_NextState()) != SES_OKAY)
                ErrorHandling(CC);
        }
    }
}

```

Initialization

Main loop

visualSTATE deduction sequence

Example of Expert API main function

```

/* RealLink task pointer */
VS_RLTASK* pTask;
/* standard context pointer */
SEM_CONTEXT* pContext;

/* initialize RealLink */
VS_RealLinkInit(&pTask);

/* Init <system name> */
/* VS_RealLinkInit(..) must have been called prior to
   initializing any VS System */
<system name>SMP_InitAll((&pContext, pTask);
while(1)
{
    /* RealLink macro */
    VS_WAIT(pContext);

    /* Standard vS */
    nEventNo = GetEventFromQueue();
    if (nEventNo != EVENT_UNDEFINED)
    {
        SMP_Deduct(pContext, nEventNo);
        while(SMP_GetOutput(pContext, &ActionNo) == SES_FOUND)
            SMP_Action(pContext, ActionNo);
        SMP_NextState(pContext);
    }
}

/* Cleans up vS */
SMP_Free(pContext);

/* Cleans up RealLink */
/* SMP_Free() must have been called for all VS Systems */
VS_RealLinkFree(pTask);

```

Note: The communication hardware must be initialized before entering the main loop.

Step 4: Implementing target-specific functions

Because both the visualSTATE API and RealLink APIs are target-independent, they contain no information on how to use the communication device of the target.

Note: All visualSTATE Systems must be located in the same task if you want to apply RealLink.

- I In order to access the communication device you must implement the following target-specific RealLink functions that are used by the visualSTATE API:

<code>Reset()</code>	Resets the target. The function will be called by the RealLink API. This function might not be necessary for your target.
<code>Transmit()</code>	Transmits one byte on the communication port or adds bytes to the buffer. The function will be called by the RealLink API. See <i>Example: Transmit function (RS232 implementation)</i> , page 214.
<code>TransmitFlush()</code>	This function must only be implemented if a buffer is used. The function should empty the transmit buffer.
<code>Receive()</code>	Must be interrupt-based. The function receives characters from the communication device. The received characters should be passed to the RealLink protocol by calling the function <code>VS_RealLinkReceive()</code> . See <i>Example: Receive function (RS232 implementation)</i> , page 215.

You can change the default names of the functions by defining the macros:

```
#define VS_RL_RESET           MyReset
#define VS_RL_TRANSMIT       MyTransmit
#define VS_RL_TRANSMIT_FLUSH MyTransmitFlush
```

TIP: IAR MakeApp can be used to automatically generate device drivers.

Examples of implementation

The functions in the following examples were implemented using the following:

Microprocessor:	ARM7 - LPC2138
Compiler:	IAR Embedded Workbench for ARM (EWARM)

Example: Transmit function (RS232 implementation)

```
#if (VS_REALLINKMODE == 1)

/* *** UART functions *** */
/* Reset is not needed for this platform */
void RealLinkReset(void)
{
}

/* Transmits one byte via UART1 */
void RealLinkTransmit(unsigned char byte)
{
    unsigned char status;

    /* Wait for TXRDY */
    do
    {
        status = U1LSR;
    }
    while ((status & 0x20) == 0);

    U1THR = byte;
}

#endif
```

Note: The function does not transmit new data until the transmit register is empty.

Example: Receive function (RS232 implementation)

```

/* Receive Interrupt routine for RealLink */
#if (VS_REALINKMODE == 1)
static void UART1Interrupt()
{
    switch(U1FCR_bit.IID)
    {
        case IIR_CTI:
        case IIR_RDA:
            VS_RealLinkReceive(U1RBR); /* Receive data available */
            /* Call received byte callback */
            /* function */

            break;
        case IIR_THRE:
            /* THRE interrupt */
        case 0x0:
            /* Modem interrupt */
        case IIR_RSL:
            /* Receive line status interrupt (RDA) */
            /* Character time out indicator interrupt (CTI) */

        default:
            break;
    }
    VICVectAddr = 0;
}
#endif

```

- 2 Include RealLink.h in the file where the Transmit() and Reset() functions are implemented.

Step 5: Completing the target source code

When the preceding steps have been completed, the target source code is ready and can be embedded in the target.

- 1 Compile and link the complete project.
- 2 Embed the source code in the target.

Step 6: Configuring the Validator for RealLink

By now the target application should be ready and embedded. The next step is to configure the RealLink connection, as follows:

- 1 Launch the Validator and open your workspace.

- 2 On the menu, choose RealLink>Properties. A RealLink Properties dialog box will be displayed. See *Figure 166*, page 216.

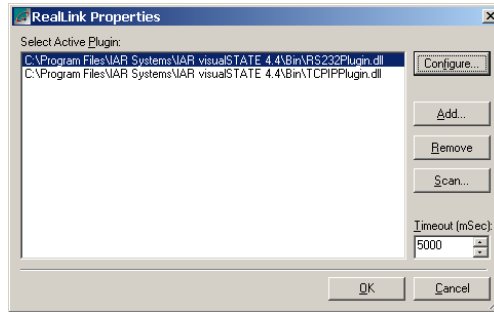


Figure 166: RealLink Properties dialog box

- 3 In the Select Active Plugin list, select the communication plugin to use, or click the **Add** button to browse for other plugins and add them to the list.
Alternatively, click the **Scan** button to scan a specific folder for communication plugins.
- 4 When you have selected a communication plugin, it must be configured to the same communication settings as those implemented on the target. Click the **Configure** button. A dialog box for setup of the selected communication plugin will be displayed. See *Setting up RealLink RS232 communication plugin*, page 217, and *Setting up RealLink TCP/IP communication*, page 218.

Clicking the **Remove** button will remove the selected communication plugin from the list.

Information about the RealLink communication plugin selected is stored in the current Validator workspace.

Setting up RealLink RS232 communication plugin

- 1 Perform the steps described in *Step 6: Configuring the Validator for RealLink*, page 215. When you click the **Configure** button after having selected RS232 communication (see *Figure 166*, page 216), the RS232 Setup dialog box will be displayed. See *Figure 167*, page 217.

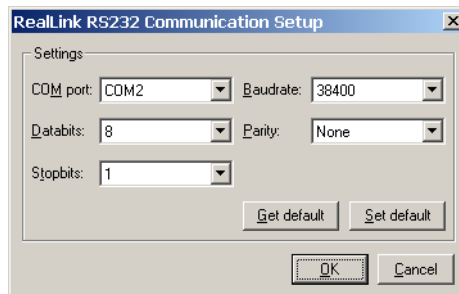


Figure 167: RS232 Setup dialog box

- 2 Fill in the dialog box with the appropriate settings which must match the target settings.

Click the **Get Default** button to apply the default settings.

Click the **Set Default** button to create a new default setting.

The visualSTATE RealLink RS232 plugin must have exclusive access to the serial port; it cannot be shared with other programs. You will get an error message if trying to open a serial port that is already in use by another program.

- 3 Go to *Establishing the first RealLink connection*, page 219.

Setting up RealLink TCP/IP communication

- 1 Perform the steps described in *Step 6: Configuring the Validator for RealLink*, page 215. When you click the **Configure** button after having selected TCP/IP communication, the Setup dialog box for TCP/IP communication will be displayed. See *Figure 168*, page 218.

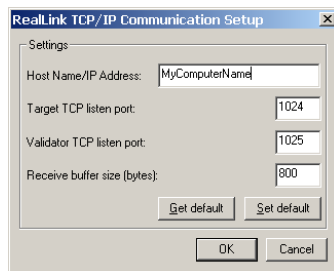


Figure 168: TCP/IP Communication Setup dialog box

- 2 In the Host Name/IP Address field, type target host name or IP address.
- 3 Specify target listen port and Validator listen port. The reason for this is that both the target and the RealLink TCP/IP communication plugin listen on a specific port in order to establish a connection to the target. By default, the following ports are used:
 - Port 1024 is used as the target listen port.
 - Port 1025 is used as the Validator listen port.
- 4 Specify size of receive buffer. The size you should specify depends on the visualSTATE model. Set receive buffer size to at least the size of the largest entity that will be transferred between the target and the Validator. This could for example be the state vector, or a variable defined as a large array. The buffer size only affects communication performance, not the functionality.

Implementation

To set up and configure RealLink in your target, see *Setting up RealLink*, page 207.

You may find it useful to add the `RL_TCPIP.cpp` file to your target project. The `RL_TCPIP.cpp` file uses the Windows Sockets API to implement the TCP/IP communication. Because the file uses the Berkeley function set to the widest possible degree, it will be relatively easy to port the `RL_TCPIP.cpp` file to other platforms.

- 5 Go to *Establishing the first RealLink connection*, page 219.

Setting up your own TCP/IP communication

If you prefer to set up your own TCP/IP communication in the target instead of using the `RL_TCPIP.cpp` file, do the following:

- 1 Set up a server to listen on the port you have configured as the target listen port. All data from the Validator will be sent to this port and any received data should be handed to the RealLink API.
- 2 Each time a connection is established on this port, extract the Validator IP address from the connection.
- 3 Using the Validator IP address, create a connection to the port you have configured as the Validator listen port. All data to be sent to the Validator should be sent via this connection. Thus the RealLink transmit function should use this connection.
- 4 Go to *Establishing the first RealLink connection*, page 219.

ESTABLISHING THE FIRST REALLINK CONNECTION

When the communication plugin has been configured as described in *Step 6: Configuring the Validator for RealLink*, page 215, you can establish a RealLink connection, as follows:

- 1 On the Validator menu, choose `RealLink>Connect` as shown in *Figure 169*, page 219.

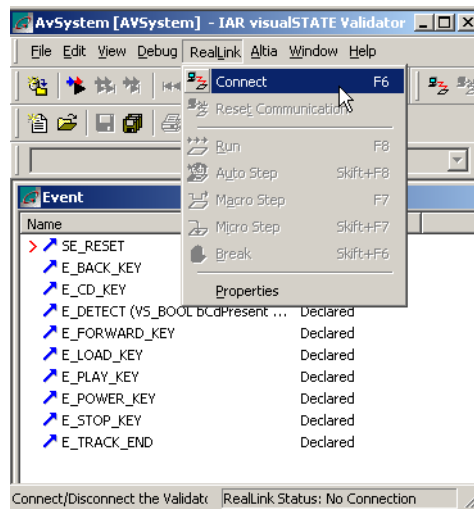


Figure 169: Connecting to RealLink

- 2 If the connection is successfully established, the Validator output window (RealLink tab) will display the message shown in *Figure 170*, page 220.

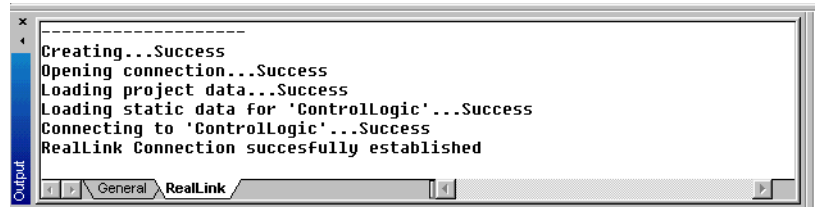


Figure 170: Validator output window

When the RealLink connection has been successfully established, the Validator will stop execution when the `VS_WAIT()` macro is reached for the first time (`VS_WAIT()` is the macro that you inserted in the target application code, see *Using the RealLink API*, page 210). `VS_WAIT()` continually checks if execution should be halted.

You can now monitor and control the target application, as described in *Monitoring your target application*, page 220 and *Controlling your application in target*, page 224.

Monitoring your target application

When you have created a visualSTATE application with support for visualSTATE RealLink, and you have established a RealLink connection (see *Setting up RealLink*, page 207), you can monitor and control the behavior of the visualSTATE model in the target.

It is possible to monitor the behavior of all visualSTATE elements in target via the Validator windows, except action function parameters and guard expressions (see *Monitoring visualSTATE elements*, page 222).

USING VALIDATOR WINDOWS WITH REALLINK

By default, all open windows in the Validator show the Validator representation of the visualSTATE model. However, when RealLink is used, the windows can be changed so as to show the status of the target model. The only window that cannot be changed to showing RealLink is the Guard Expression window. Generally, the windows in target mode correspond to the windows in Validator mode (see *Validator windows*, page 152).

The title bars of the windows show which version of the model is shown: Validator model or target model (runtime model). See *Figure 171*, page 221.

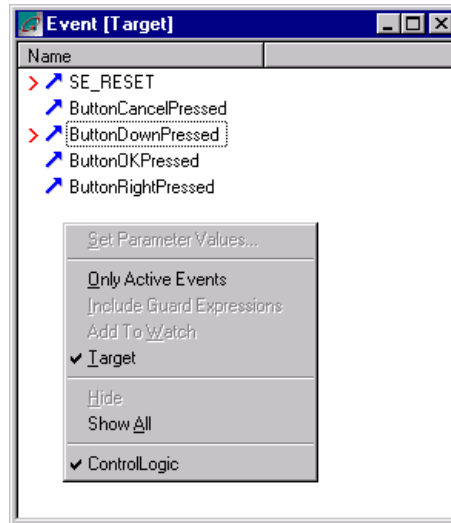


Figure 171: Validator Event window in target mode

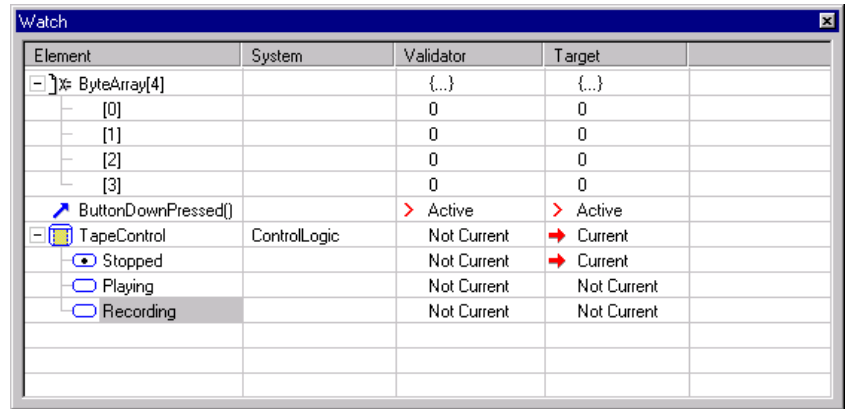
The Validator keeps track of which windows are set to target mode, and will automatically open them next time ReaLink is connected.

To change between Validator mode and target mode in Validator windows:

- 1 Select the appropriate window, for example Event window.
- 2 Open the pop-up menu and choose **Target** to toggle the mode, or press ALT + F8. See *Figure 171*, page 221.

Validator Watch window

For monitoring visualSTATE elements, you can also use the Validator Watch window. You can add the various types of visualSTATE elements to this window, which shows both the Validator model and the target model. See *Figure 172*, page 222.



Element	System	Validator	Target
[-] % ByteArray[4]		{...}	{...}
[-] [0]		0	0
[-] [1]		0	0
[-] [2]		0	0
[-] [3]		0	0
[+] ButtonDownPressed[]		> Active	> Active
[-] [x] TapeControl	ControlLogic	Not Current	➔ Current
[-] [x] Stopped		Not Current	➔ Current
[-] [x] Playing		Not Current	Not Current
[-] [x] Recording		Not Current	Not Current

Figure 172: Validator Watch window containing visualSTATE elements

MONITORING VISUALSTATE ELEMENTS

In target mode, simulation is performed in the same way as in Validator mode (see *Simulation*, page 161).

The following visualSTATE elements can be monitored via the Validator windows:

Events	In the Event window, it is possible to see whether an event is active or not. If an event is active, it will be marked with a red arrow. The evaluation of whether or not an event is active is actually performed in target using the visualSTATE Basic API and Expert API. This means that the value of guard expressions is not considered, and if your target application does not include the <code>SEM_Inquiry()</code> / <code>SEM_GetInput()</code> functions, all events will be marked as being active.
Event parameters	In the Event window, you can see the values of event parameters used the last time a deduction with a specific event was performed, or the value you have set.
Variables	In the Variable window, you can see the value of both external and internal variables.

	<p>TIP: If only a single element of an array is of interest, select this element in the Variable window and press SHIFT + F9 to show the element in the Watch window.</p>
System state	<p>In the System window, you can monitor the current state of a System. If a state is currently active, it is marked with a red arrow.</p> <p>Graphical animation (Debug>Graphical Animation) is also available when using Reallink. By using this option you can monitor the current states in the statecharts of visualSTATE Designer.</p> <p>TIP: If only a single branch of a System is of interest, select the branch in the System window. Then either press SHIFT + F9 to show the branch in the Watch window, or choose the New Branch command from the pop-up menu to add the branch to the System window as a separate branch.</p>
Signal queue	<p>The Signal Queue window shows the signal queue of all Systems.</p>
Executed actions	<p>The Action window lists the actions executed during the last step. This includes both executed action functions and assignments.</p>

MANIPULATING YOUR TARGET APPLICATION FROM WITHIN THE VALIDATOR

When the `VS_WAIT()` macro is reached and execution of your target application stops, you have the following options of manipulating it from within the Validator:

- Changing variable values
- Sending events into target.

Changing variable values

The value of a variable can be changed either in the Variable window or in the Watch window. See *Figure 173*, page 223.

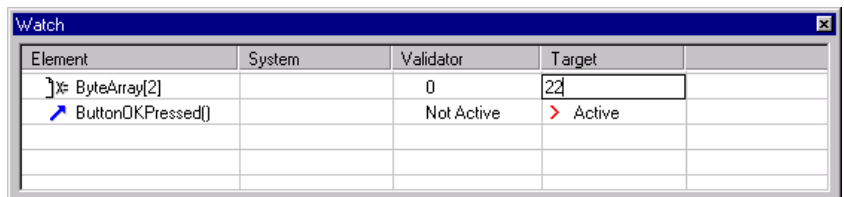


Figure 173: Editing a variable in the Watch window

Sending events into target

When you double-click on an event in the Event window (or select an event in the Watch window and press ENTER), the event will be sent into the target and processed just as if the event had occurred, for example due to a button being pressed.

Note: An event sent from the Validator bypasses all event queues in the target.

If the event has parameters, the Validator holds a copy of the values of these parameters. Between deductions, the Validator event parameter values are shown. Until the first deduction, the event parameter values are undefined.

Values can be assigned to event parameters in either of the following ways:

- If an event that occurred in target is processed and the event is shown either in an Event window in target mode, or in the Watch window, then the Validator event parameters will be assigned the value that the target event parameters have during the processing.
- Alternatively, event parameters can be assigned a value in the Watch window (see *Watch window*, page 157).

Note: In Autostep mode and Run mode, you cannot send events into the target model.

Controlling your application in target

It is possible to break execution of code in target. Breaks are performed on the following two macros:

- `VS_WAIT()` macro which you must insert manually in the main loop (see *Setting up RealLink*, page 207). When `VS_WAIT()` is reached, the Validator exchanges data with the runtime application and updates all logical elements, according to the options selected.
- A macro in the visualSTATE API which is parallel to `VS_WAIT()`.

Break by the `VS_WAIT()` macro corresponds to break on a *macrostep*. Break on the parallel macro in the visualSTATE API corresponds to break on a *microstep*.

MICROSTEPS AND MACROSTEPS

The visualSTATE concepts of microsteps and macrosteps are related to event processing and signal queue handling. A *macrostep* includes the processing of the event and furthermore handling of any signals added to the queue as a result of the event processing. A *microstep* is the processing of a single event or signal, and thus a

macrostep comprises at least one microstep. In visualSTATE, the signal queue is completely emptied before a macrostep is finished (see *Figure 174*, page 225).

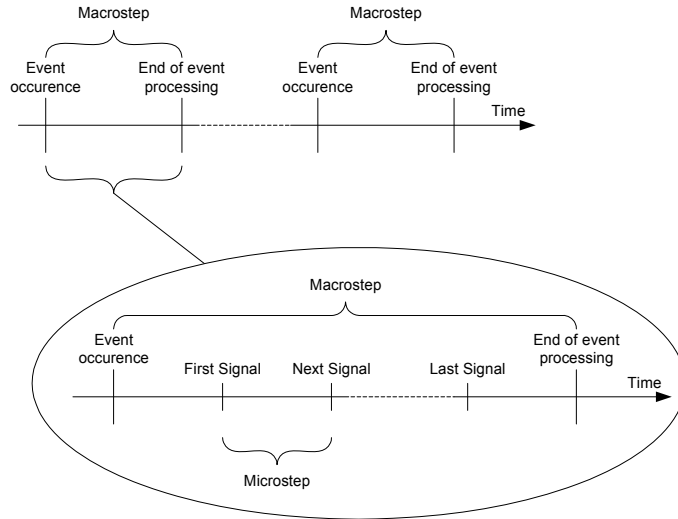


Figure 174: Microstep and macrostep in visualSTATE

For more information on signal queues and event processing in visualSTATE, see *IAR visualSTATE Reference Guide*.

CONTROLLING EXECUTION OF CODE IN TARGET

Immediately after the Reallink connection with the target has been established, the Validator will try to stop execution of the code when the first instance of the `VS_WAIT()` macro is reached. When code execution stops, you can use the Reallink menu commands to continue execution and thereby debug your application. The Reallink

menu commands are shown in *Figure 175*, page 226 and explained in *Commands for controlling execution of code in target*, page 226.

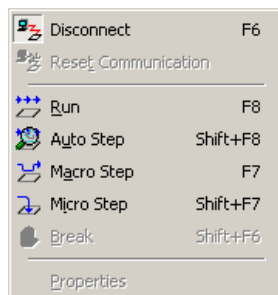


Figure 175: Validator RealLink menu commands

Commands for controlling execution of code in target

For controlling execution of code in target, the following RealLink commands are available on the Validator menu:

- Microstep** When you choose this command, a deduction with the next trigger will be performed. In other words, execution will continue until either the `VS_WAIT ()` macro, *or* the parallel microstep macro in the visualSTATE API is reached.
- If the starting point for the **Microstep** command is that execution is stopped on a microstep (the microstep macro), it means that there are signals in the signal queue. Thus a deduction will be performed using the first signal in the queue.
- If the starting point for the **Microstep** command is that execution is stopped on a macrostep (the `VS_WAIT ()` macro), a deduction will be performed using the next event in the event queue. This results in one of the following cases:
- If no events exist in the queue, this corresponds to one loop in the visualSTATE main loop, without any deduction being performed.
- If an event is processed, and this results in signals being added to the queue, execution will stop before processing the first signal (microstep macro). This corresponds to break on a microstep.
- If an event is processed, and no signals are added to the queue, execution will stop upon the next occurrence of the `VS_WAIT ()` macro. This corresponds to break on a macrostep.

	<p>For a description of microsteps and macrosteps, see <i>Microsteps and macrosteps</i>, page 224.</p>
Macrostep	<p>When you choose this command, execution will continue until the <code>VS_WAIT()</code> macro is reached.</p> <p>If the starting point for the Macrostep command is that execution is stopped on a microstep (the microstep macro), it means that there are signals in the signal queue, and processing will be performed with the first signal. If the queue still holds signals, processing with the next signal will be performed. This continues until the signal queue is empty, and the <code>VS_WAIT()</code> macro is reached.</p> <p>If the starting point for the Macrostep command is that execution is stopped on a macrostep (the <code>VS_WAIT()</code> macro), processing with the next event in the event queue will be performed. If processing of this event results in signals being added to the queue, processing is continued until the entire queue has been emptied, and the <code>VS_WAIT()</code> macro is reached again. As with the microstep, if no events exist in the queue, this corresponds to one loop in the visualSTATE main loop, without any processing being performed.</p> <p>For a description of microsteps and macrosteps, see <i>Microsteps and macrosteps</i>, page 224.</p>
Autostep	<p>Choosing this Reallink command allows you to continuously execute the code in target, while at the same time monitoring the values of the visualSTATE elements. Each time a microstep or macrostep is reached, the values of the visualSTATE elements will be updated. When the values have been updated, execution in target will continue.</p>
Run	<p>Choosing this command lets the target execute code at a speed as close as possible to that of a non-RealLink application. The only difference in speed between this mode and a non-RealLink application is that each time one of the break macros are passed, for example <code>VS_WAIT()</code>, the target checks whether or not it should stop execution. Note that if Record is turned on (<code>Debug>Record</code>), Run mode corresponds to Autostep mode because the values of all visualSTATE elements are needed for the test sequence file.</p>

Recording and playing sequences of target tests

The commands for recording and playing test sequences are also available when the Validator is connected to a target. This means that it is possible both to record a sequence executed in target, and to play a previously recorded sequence in target by means of a

test sequence file. A sequence recorded in target can also be used as input to a dynamic analysis in order to see the test coverage.

For a description of how to use test sequence files, see *Recording and playing test sequences*, page 187.

Troubleshooting

If RealLink fails to connect to the target microcontroller, a message box like the one shown in *Figure 176*, page 228 will appear (message box text depends on the specific error).

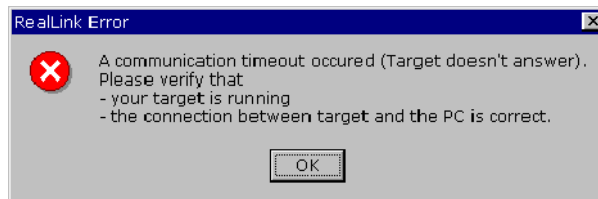


Figure 176: RealLink communication error message

The message box appears when the Validator has transmitted data to the microcontroller and does not receive any valid response from the target after a number of seconds. If you receive this error message, check the following:

GENERAL

- Is the cable between the PC and the target microcontroller connected?
- Is the target microcontroller powered on?
- Is the target software loaded and started?
- Is the correct communication plugin selected? See *Configuring the Validator for RealLink*, page 215.
- Is the communication plugin correctly configured—does it match the target settings? See *Setting up RealLink*, page 207.
- Is the cable between the PC and target microcontroller very long, or is there much electronic noise in the environment? If so, try lowering the baud rate in both the Validator and the microcontroller.
- Are the `RealLinkTransmit()` and `RealLinkReceive()` functions working?
TIP: Use a terminal program to transmit a known value to the microcontroller and have it echo it back. For example use HyperTerminal that ships with Microsoft Windows 9x/NT/2000

SETTINGS FOR RS232 COMMUNICATION PLUGIN

- Are the baud rate, data bit, stop bit, parity, and hardware handshaking correct? If not, change the communication settings in the Validator to match the settings in the microcontroller, as explained in *Configuring the Validator for Reallink*, page 215.
- Is another program using the serial port? If so, close the other program using the serial port. Other programs using a serial port include modem software, PDA synchronization software, etc.

DIGITAL SIGNATURE

- Is the statechart loaded in the Validator the same as the one running in the target microcontroller? If not, load the correct statechart into the Validator.
- Has the statechart been changed and the new program not downloaded to the target microcontroller? If this is the case, code-generate the visualSTATE model, build the complete application, and download it to the microcontroller.

Part 7: Code generation

This part of the visualSTATE[®] User Guide includes the following chapters:

- Introduction
- Generating code
- Basic API code generation
- Expert API code generation
- Size of generated code.





Introduction

On the basis of designs created with visualSTATE Designer, it is possible to automatically generate code for visualSTATE Projects (in the following referred to as *VS Projects*) by means of visualSTATE Coder.

This chapter gives an introduction to the visualSTATE Coder, and describes the following:

- Code generation and visualSTATE APIs
- Description of generated code
- Elements supported by the Coder
- Real-time operating system (RTOS).

Code generation and visualSTATE APIs

The Coder can generate code for the visualSTATE Basic APIs and visualSTATE Expert API (see *Basic API code generation*, page 239, and *Expert API code generation*, page 245). A detailed description of the visualSTATE APIs is found in *visualSTATE API Guide*.

The Coder will code generate code for one VS Project at a time, including all VS Systems and VS Statecharts of the VS Project.

Description of generated code

The Coder will generate the complete code for the visualSTATE global layer and the visualSTATE local layer. For the Basic API, it will also generate the API layer. See *Figure 177*, page 234.

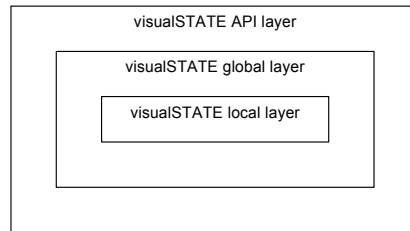


Figure 177: visualSTATE layers

visualSTATE API layer

The visualSTATE API layer is the functions used to access the state machine engine and model in runtime. The API files for the Basic API are generated at the same time as the code is generated for the global and local layer.

Note: The generation of API files for the Basic API is new from visualSTATE version 5.3.

Files for the Expert API are static and are placed in the `API` subdirectory of the visualSTATE installation.

visualSTATE global layer

The visualSTATE global layer contains what you could term *external logic*. It is external in the sense that the user interfacing to the model can access the data in some way, for example by calling an API function. The global layer includes events, constants, external variables, action expressions, and element explanations.

visualSTATE local layer

The visualSTATE local layer contains the logic that is used internally in the model. Thus it cannot be seen by the user interfacing to the model. The local layer includes transitions, guard expressions, internal variables, and signals.

CODER REPORT FILE

The Coder can generate a report file during code generation. The report file contains the following information:

For the VS Project

- Coder options
- Model characteristics
- Generated statistics.

For each VS System

- Coder options
- Model characteristics
- Generated statistics.

Summary information

- Information about the overall content of the generated code.
- Number of errors and warnings detected during code generation.

ELEMENTS SUPPORTED BY THE CODER

All elements of a VS System are supported in the visualSTATE Coder.

Real-time operating system (RTOS)

Runtime applications developed with visualSTATE can be used with or without a real-time operating system. If you choose to use OSEK as operating system, you can use the visualSTATE OSEK Kit. See *Part 10: Working in an OSEK environment*, page 311.

Generating code

This chapter describes how to start code generation from the Navigator. It also describes the options you must specify to have C++ code generated.

The code generation process can also be started from the command line, as described in *Coder options*, page 375.

For information about code generation for RealLink, see *Enabling RealLink support*, page 207.

Starting code generation

To start code generation:

- 1 Launch the Navigator, and open your workspace file.
- 2 Check that the correct visualSTATE Project is set as active and set the appropriate Coder options (see *Setting Verificator, Coder and Documenter options*, page 29).
- 3 On the menu, choose Project>Code generate to generate code for the Project.

Code generation will start, and progress will be written to the Navigator output window.

By default, the Coder-generated files are located in the `CODER` directory of the directory where your visualSTATE Project file is located. You can also specify a file output directory under the File Output tab of the Coder Options dialog box (see *Figure 21*, page 30).

Generating C++ code

To have your visualSTATE Project code generated as C++ code, do the following:

- 1 Launch the Navigator, and open your workspace file.
- 2 Check that the correct visualSTATE Project has been set as active.

- On the menu, choose Project>Options>Code generation. The Coder Options dialog box is displayed. See *Figure 178*, page 238.

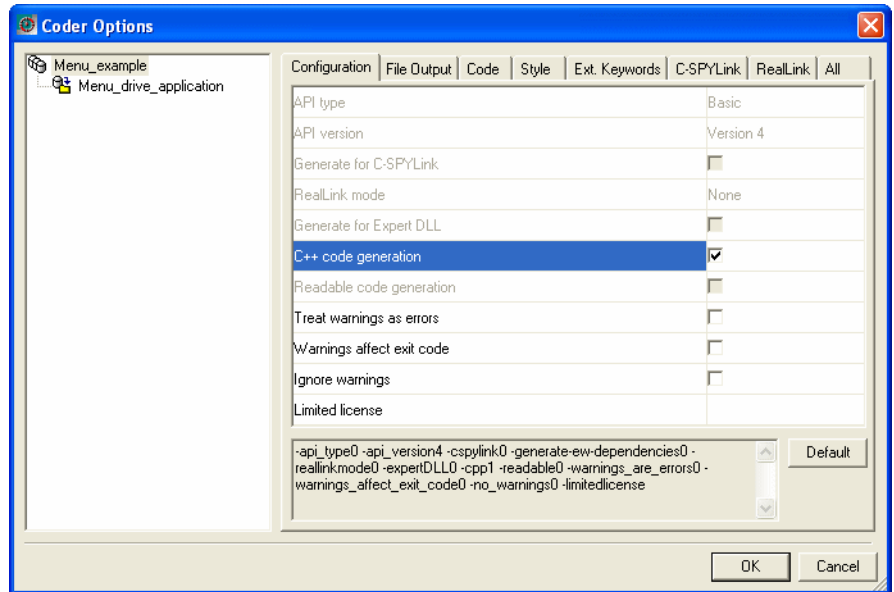


Figure 178: Navigator, Coder Options dialog box, Configuration tab

- On the Configuration tab, select **C++ code generation**.

This will set the following values that are required for C++ code generation:

API type: Basic

API version: Version 4/5

Generate for C-SPYLink: Not selected

RealLink mode: None

Generate for Expert DLL: Not selected

Readable code generation: Not selected

Internal variable initialization: By function

Functional expression handling: Switch-case construct

In addition, options for specifying extended keywords that are used for non-static members of the generated class are disabled (internal variables, double buffer data, etc.).

- On the menu, choose Project>Code generate to generate C++ code for the Project.

Basic API code generation

With the visualSTATE Basic API, code generation will be executed for one VS Project containing one or more VS Systems. In a project with more than one system, the code generated for each system is stand-alone and independent of the other Systems in the project.

Detailed information about the Basic API can be found in *visualSTATE API Guide*.

Description of generated code

Choose between two fundamentally different types of C/C++ code output:

- *Table-based code* (C or C++) for maximum compactness (as in previous versions of visualSTATE)
- *Human-readable code* (C only), a representation of the state machine logic based on `switch` and `if` statements.

The human-readable code option is useful if, for example, you are required to show traceability between high-level functional requirements and generated code. Moreover, if speed is a more critical factor than code size, human-readable code is generally preferable.

GENERATING TABLE-BASED CODE

During the code generation phase, a set of files is generated by default:

- VS Project-specific files
- VS System-specific files.

If you choose to enable *extended configuration*, the generated code is the same but it is partitioned across a set of additional VS System-specific files, for more fine-grained dependency control in your compiler project.

`<SystemName>` denotes the optional prefix used by the code generator, to distinguish files from different systems. The default prefix is the system name, but this can be changed in the **Coder Options** dialog box.

A group of files from one System can be compiled to be used by themselves in an application binary file or together with files from another system.

VS Project-specific files

<code><Project>PExtVar.h</code>	Contains the declarations of all external variables that are defined at project level and shared for all systems. This file will only be generated if needed.
<code><Project>PExtVar.c</code>	Contains the definitions of all external variables that are defined at project level and shared for all systems. This file will only be generated if needed.

VS System-specific files

<code><System>.c</code>	Contains the core model logic of the VS System (primarily transitions). This file is part of the local layer in <i>Figure 177</i> , page 234.
<code><System>.h</code>	Header file for <code><System>.c</code>
<code><System>Data.c</code>	Contains additional logic for the VS System (primarily guard expressions, action expressions, and variables).
<code><System>Data.h</code>	Header file for <code><System>Data.c</code>
<code><System>Action.h</code>	Contains the external declarations of action functions and action expressions in the VS Project and VS System.
<code><System>SEMLibB.h</code>	Contains the function definitions for the API functions.
<code><System>SEMLib.h</code>	Contains the function declarations for the API functions.
<code><System>SEMTypes.h</code>	Contains a set of Coder-generated type definitions named SEM type definitions.
<code><System>SEMDef.h</code>	Contains macro definitions and type definitions that configure the Basic API.

VS System-specific files, generated with extended configuration

Note: The Extended configuration is enabled by specifying file names for *Action function header file*, *External variable header file*, *External variable source file*, and *Constant header file* on the **File Output** page of the **Coder Options** dialog box.

<code><func>.h</code>	Contains external declarations of action functions. By default the declarations are located in the <code><System>Action.h</code> file.
-----------------------------	--

<ext>.c Contains the external variables of the VS System (and of the VS Project). The filename for project-external variables will have a default value and the file cannot be removed, but it is only generated if it is needed.

By default the external variables for a System are located in the <System>Data.c file.

<ext>.h Header file for <ext>.c.

<constant>.h Contains the VS Project and VS System constants. By default the constants are located in the <System>Data.h file.

GENERATING HUMAN-READABLE CODE

The human-readable code option can only generate C code and the resulting application cannot be debugged using RealLink. It can, however, be debugged with the C-SPY Simulator or hardware emulator debugger in IAR Embedded Workbench, using C-SPYLink. The human-readable code option is set on the project level on the **Configurations** page.

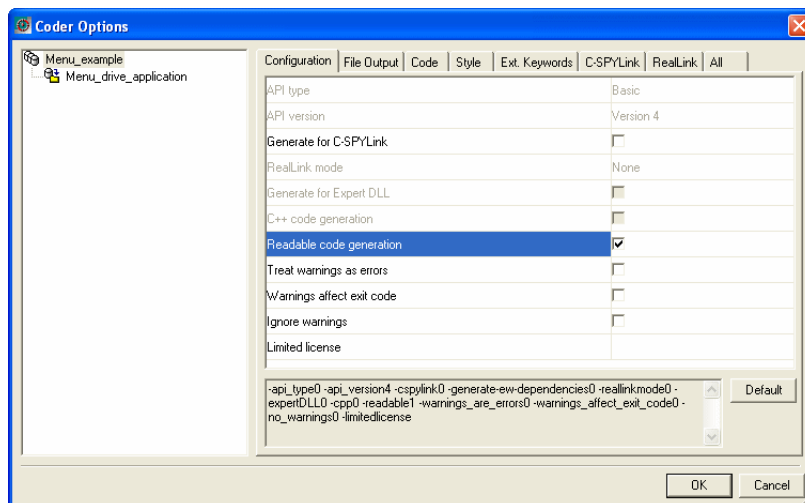


Figure 179: Enabling human-readable code generation

If you use the human-readable code option, both the API for calling the generated code and the set of generated files are simplified. These files are generated:

```
<System>SEMTypes.h  
<System>SEMLib.h  
<System>SEMLib.c  
<System>SEMDef.h
```

Declarations of action functions as well as declarations and definitions of internal and external data are included in the `<System>SEMLib.h` and `<System>SEMLib.c` files. To access the declarations, include the `<System>SEMLib.h` file in your own files.

Furthermore, the table-based code API functions `SEM_Deduct`, `SEM_GetOutput`, `SEM_Action`, and `SEM_NextState` are replaced by a single function call to `<System>VSDeduct` that accepts the same parameters as `SEM_Deduct`. `<System>VSDeduct` calls all action functions and action expression, and changes the state appropriately.

Default table-based code configuration

This figure shows the Coder-generated files and Basic API files that are used in a Basic API default configuration for table-based code.

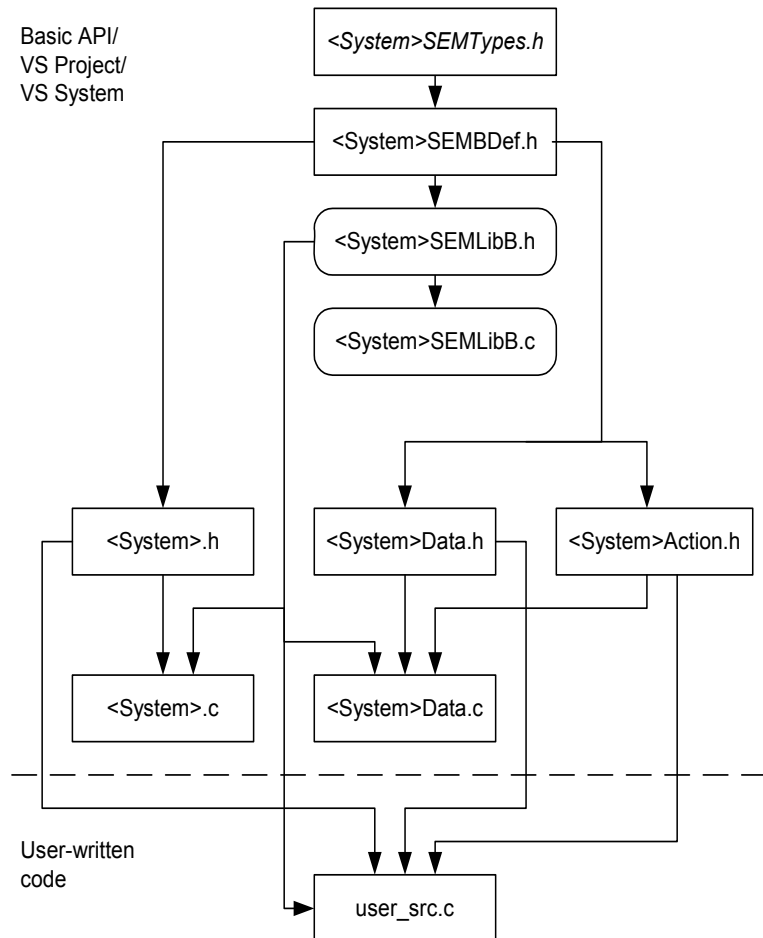


Figure 180: Basic API, default configuration

Note to Figure 180: Rectangles with rounded corners represent the source and header files that are part of the visualSTATE API. The arrows in the figure indicate how the header files are included in the source files.

Expert API code generation

This chapter describes code generation with visualSTATE Expert API. With this API, code generation will be executed for a VS Project containing one or more VS Systems at a time.

Detailed information about the Expert API can be found in *visualSTATE API Guide*.

Description of generated code

During the code generation phase, the following set of files is generated by default:

- VS Project-specific files
- VS System-specific files (for each VS System).

If you choose extended configuration, a set of additional VS System-specific files are also generated.

VS Project-specific files

SEMTypes.h	Contains a set of Coder-generated type definitions named SEM type definitions.
SEMEDef.h	Contains macro definitions and type definitions that configure the Expert API.
<gcext>.c	Contains the external variables of the VS Project.
<ghext>.h	Header file for <gcext>.c.
<gconstant>.h	Contains the constants of the VS Project.
<gevent>.h	Contains the events of the VS Project.

VS System-specific files (for each VS System)

<source>.c	Contains the core model logic of the VS System (primarily transitions). This file is part of the local layer in <i>Figure 177</i> , page 234.
<header>.h	Header file for <source>.c.

<code><sdata>.c</code>	Contains additional logic for the VS System (primarily guard expressions, action expressions, variables).
<code><hdata>.h</code>	Header file for <code><sdata>.c</code> .
<code><action>.h</code>	Contains the external declarations of action functions and action expressions in the VS Project and VS System.

VS System-specific files, generated with extended configuration (for each VS System)

<code><func>.h</code>	Contains external declarations of action functions. By default the declarations are located in the <code><action>.h</code> file.
<code><cext>.c</code>	Contains the external variables of the VS System. By default the external variables are located in the <code><sdata>.c</code> file.
<code><hext>.h</code>	Header file for <code><cext>.c</code> .
<code><constant>.h</code>	Contains the VS System constants. By default these are located in the <code><sdata>.h</code> file.

Default configuration

Figure 181, page 247 shows the Coder-generated files and Expert API files that are used in an Expert API default configuration.

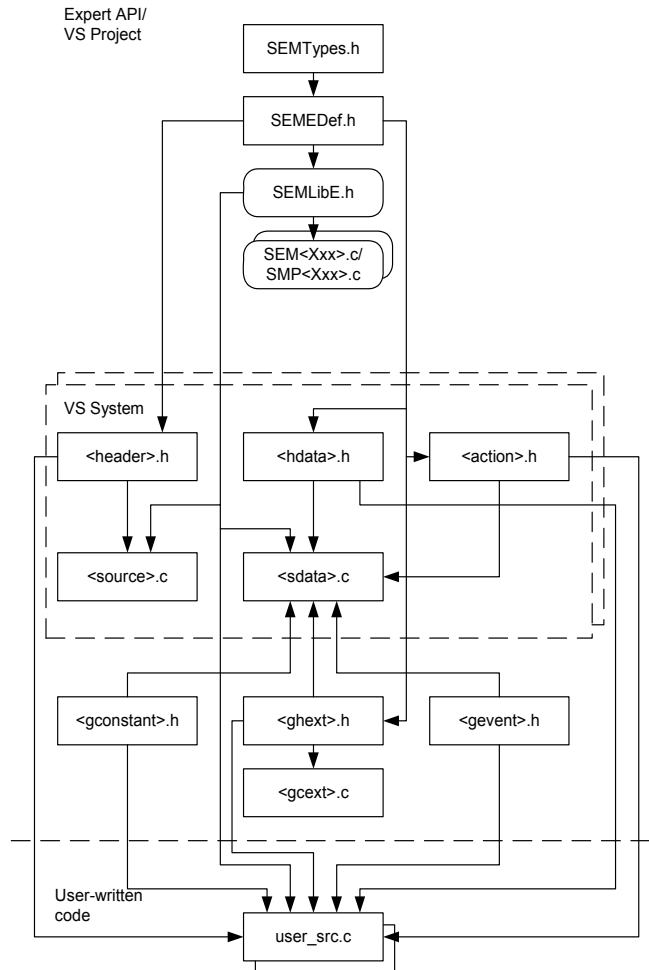


Figure 181: Expert API, default configuration

Note to *Figure 181*, page 247: Rectangles with rounded corners represent the source and header files that are part of the visualSTATE API. The arrows in the figure indicate how the header files are included in the source files.

Size of generated code

The size of the generated code depends on the data width and rule data format applied by the Coder. By default, the Coder will optimize for size.

This chapter describes how data width and rule data formats influence the size of generated table-based code. At the end, the size of human-readable code is discussed.

Data width

The data width determines the size of SEM type definitions (see *Table 6*, page 249). The size of the individual SEM type definitions can be 8 bit, 16 bit or 32 bit.

During the generation of table-based code, the Coder will by default optimize the size of each SEM type definition. However, it is possible to force all SEM type definitions to be of the same width by setting the data width to either 8, 16 or 32 bit. In this case all SEM type definitions will have the same width.

The Coder option `-D` determines the VS System data width for all SEM type definitions.

Type identifier	Explanation
SEM_EVENT_TYPE	Event variable type.
SEM_EVENT_GROUP_TYPE	Event group variable type.
SEM_GUARD_EXPRESION_TYPE	Used internally in visualSTATE APIs.
SEM_STATE_TYPE	State variable type.
SEM_ACTION_FUNCTION_TYPE	Action function variable type. Used only for action functions without parameters and which have the return type VS_VOID.
SEM_ACTION_EXPRESSION_TYPE	Action expression variable type.
SEM_SIGNAL_QUEUE_TYPE	Signal queue variable type
SEM_INSTANCE_TYPE	Instance variable type.
SEM_STATE_MACHINE_TYPE	State machine variable type.
SEM_EXPLANATION_TYPE	Explanation variable type.
SEM_INTERNAL_TYPE	Used internally in visualSTATE APIs.
SEM_RULE_INDEX_TYPE	Used internally in visualSTATE APIs.
SEM_RULE_TABLE_INDEX_TYPE	Used internally in visualSTATE APIs.

Table 6: Coder-generated SEM type definitions

Type identifier (Continued)	Explanation
SEM_EGTI_TYPE	Used internally in visualSTATE APIs.

Table 6: Coder-generated SEM type definitions

TIP: The SEM type definitions will be defined in the Coder-generated file `SEMTypes.h`. SEM type definitions are defined by using either the `typedef` keyword or the `#define` keyword. Use the Coder option `-tsem` to specify the keyword to use.

Rule data formats

Table 7, page 250 shows the rule data header word type, rule data header word width, and rule data width of the different rule data formats. By default, the Coder will optimize the size of the rule data format number.

The rule data format is used for storing transitions in the visualSTATE local layer. Each transition consists of one rule data header word and one rule data element (see Table 7, page 250) for each element of the transition (guard expression, state condition, signal, etc.).

For VS Projects that do not use guard expressions and/or signals, you can apply rule data formats with all data header types (type 1, 2, or 3). For VS Projects that contain guard expressions and/or signals, you must apply a rule data format with rule data header word type 2 or 3.

Rule data format number	Rule data header word type	Rule data header word width	Rule data width
0	Type 1	16 bit	8 bit
1	Type 2	24 bit	8 bit
2	Type 1	32 bit	8 bit
3	Type 2	48 bit	8 bit
4	Type 1	16 bit	16 bit
5	Type 3	32 bit	16 bit
6	Type 1	32 bit	16 bit
7	Type 2	48 bit	16 bit
8	Type 1	32 bit	32 bit
9	Type 3	64 bit	32 bit

Table 7: Rule data formats

It is always possible to force the Coder to use a larger rule data format than the rule data format determined by the Coder as the optimum one.

The Coder option `-rdm` determines the rule data format to be used.

TIP: The rule data format number 4 is compatible with visualSTATE Classic version 3, provided the VS System data width has been set to 16 bit. The rule data format number 8 is compatible with visualSTATE Pro version 3, provided the VS System data width has been set to 32 bit.

Coder options

See *Table 36*, page 379, for reference information about the functional expression handling coder option (`-funcexph`) and other code project options.

Code size using visualSTATE

Contrary to popular belief, automatic code generation from design models does not automatically lead to a huge code size overhead.

EXECUTION ENGINE OVERHEAD

visualSTATE® can generate code from UML state machine diagrams, using a table-driven approach. The tables are generated in a way that is extremely compact, but requires a runtime execution engine. This is common to all table-driven solutions and is not limited to state machines.

The execution engine represents a fixed overhead in terms of code size. However, this overhead is extremely small when used with a modern compiler. Since the code generated from the model is so tight, the advantage over hand-coding the model is apparent even for small state machines.

visualSTATE can also be configured to generate human-readable C code, see *Generating human-readable code*, page 241.

THE CODE

A fully implemented visualSTATE® application consists of the following parts:

- The actual application using the state machine(s)
 - This includes all startup code and generic runtime library code as used by the particular target hardware and compiler.
- The API file for the execution engine. (`SEMLibB.c`)
- The generated code – typically split in a number of files. The code consists of:
 - The state machine tables
 - Variables and expressions defined in the model
 - Declarations of action functions
 - Definitions of action expression functions.

- Action functions implemented by you and called by the state machine.

All these parts are combined to give the footprint of the complete application. visualSTATE only determines the size of the API and the generated code; the other parts are fully controlled by you and are more or less independent of the implementation model for the state machine.

A typical visualSTATE application uses a limited set of the functions present in the API to insert stimuli into the state machine and process input.

THE SIZE

To measure the *minimum* size of the API code, a minimal state machine can be created and compiled. The model consists of an initial state, a simple state and a default transition that also assigns to an externally defined variable. The API functions used are the ones typically used by a visualSTATE application. (Most other functions available in the API are for advanced use to enable very fine-grained control of the state machine or for debugging purposes.)

This type of minimal application was compiled with five different IAR Systems compilers, two 8-bit products, two 16-bit products, and one 32-bit product. All of them are built on the latest technology platform. The compilations were performed at optimization level `-z9`, the highest level of size optimization. No target-specific tuning was applied.

To find the *maximum* size of the API, a different method was used. The visualSTATE code generator configures the API to exclude internal functionality that is not needed for a particular model. This is dependent on the usage of specific model constructs, like guard expressions, state conditions, action functions, action expressions, signals, etc. To measure the API for a realistic application, a model was created that utilizes all of these model constructs. It is sufficient to use the construct once in the model, to activate the associated runtime code.

Results

For a modern compiler and a modern code generator, the overhead associated with automatic code generation is small. Of the 5 compilers that were tried, 4 had API sizes between 256 and 748 bytes, depending on the set of modeling constructs that was used. This is a small price to pay for the benefits of high level design, verification, and test.

The size of human-readable code

The size of human-readable code is harder to calculate in advance than the size of table-based code.

The number of transitions affects the code size, because each guard expression, assignment, and action function call on a transition is generated "inline" in the generated state machine logic. (In table-based code generation, calls of actions and guards are handled by fixed API code.)

The code size is also affected by the contradiction test (or ambiguity conflict test) that is generated for each transition. However, for human-readable code, this test code is not generated for transitions where it is trivial for the code generator to detect that there can be no transition contradictions. To turn off the generation of contradiction test code, see Table 44, *Code system options*, page 387.

Moreover, human-readable code is much more dependent on the target compiler behavior than table-based code. In table-based code generation, the data needed to represent the model is fixed and cannot be influenced by the compiler, except for minor alignment issues and similar things.

Note: If you generate human-readable code and use the Split readable code option, the code size increases slightly. See Table 45, page 389.

Part 8: Documenting visualSTATE Projects

This part of the visualSTATE[®] User Guide includes the following chapter:

- Introduction
- Setting up a visualSTATE Project report.





Introduction

For documentation of your visualSTATE Projects, you can create customized reports by means of the visualSTATE Documenter. The Documenter can be activated via the Navigator or the command line.

This chapter describes

- The generated Project report.
- How to create a Project report by means of the visualSTATE Navigator.
- How to view the generated Project report.

For information on using the command line for creation of Project reports, see *Documenter options*, page 393.

Project report

A visualSTATE Project report generated by the Documenter includes information on design, functional and formal testing, generated code and implementation of your visualSTATE Project. All relevant Project information is collected from the other visualSTATE modules and organized into a structured document. The document can be in HTML format, or RTF (rich text format), according to your selection.

The information in the Project report is based on a number of visualSTATE files, as shown in *Figure 182*, page 258.

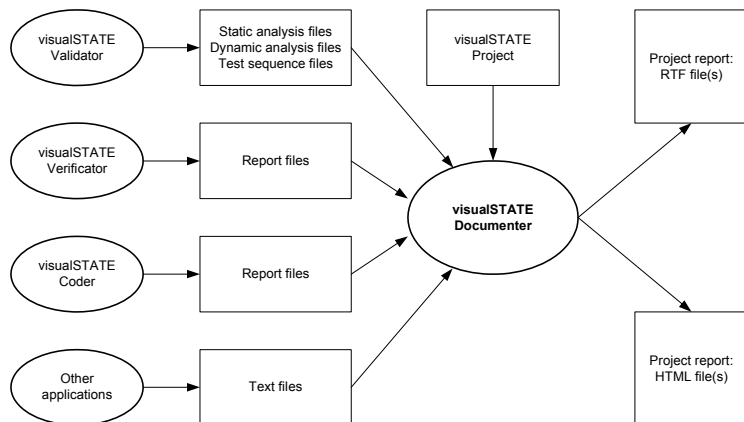


Figure 182: Files that can be included in a visualSTATE Project report

INFORMATION IN GENERATED REPORT

It is possible to specify which information should be included in the report, for example design and test, just as you can also choose between various levels of detail for the report. See *Setting up a visualSTATE Project report*, page 261.

Creating a Project report

To create a visualSTATE Project report:

- 1 Launch the Navigator and open your workspace file.
- 2 In the workspace view of the browser, select the visualSTATE Project for which to create a report.
- 3 On the menu, choose Project>Document. Report generation will start, and progress information will be written to the Navigator output window (Document tab).

The generated report is displayed in the HTML viewer of the Navigator, and a reports folder containing the report is created in the browser.

Note: If you have opened a generated Project report (file name extension `rtf`) in Microsoft Word, close the RTF file before you start creating a new Project report in RTF. For some systems it may also be necessary to close the Word application.

To change settings for report generation, see *Setting up a visualSTATE Project report*, page 261.

Viewing the Project report

By default, reports generated by the Documenter are located in a subdirectory named `Doc` in the directory containing your visualSTATE Project file.

Project reports in RTF format can be opened with for example Microsoft Word.

Note: If you open the RTF file in Microsoft Word, you will probably find that the table of contents is not updated. Update the table of contents by right-clicking on it and choose **Update Field** from the pop-up menu.

To update the page references in the entire RTF document, press CTRL+A to select all, and press F9 to update all fields.

Setting up a visualSTATE Project report

When creating a Project report, you can choose the default settings, or you can customize it in various ways. To customize the Project report, you set a number of Documenter options. Documenter options can be set in the Navigator, as described here, or using the command line (see *Documenter options*, page 393).

This chapter addresses the following issues related to setting up a Project report:

- Specifying report contents
- Specifying report output format
- Setting up standard report layout
- Customizing report layout

General

You set options for your report in the Documenter Options dialog box of the Navigator as follows:

- 1 Launch the Navigator, and open your workspace file.
- 2 In the workspace view of the Navigator browser, click the Project for which to specify Documenter options.
- 3 Open the pop-up menu, and choose Options>Documentation. The Documenter Options dialog box is displayed. Selected values are shown as command line options in the pane below the option list. See *Figure 183*, page 262.

ONLINE HELP

To view the online help for an option, right-click the option in the options dialog box, or press SHIFT+F1.

Specifying report contents

To specify the contents of the report, you select sections to include, detail level and files to use as input.

SPECIFYING SECTIONS AND DETAIL LEVEL OF REPORT

- 1 Open the Documenter Options dialog box (as described in *General*, page 261), and click the Configuration tab. See *Figure 183*, page 262.

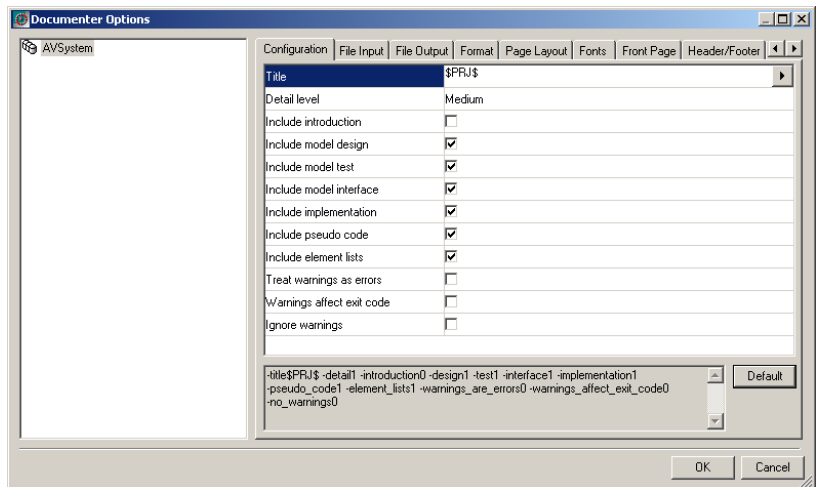


Figure 183: Documenter Options dialog box, Configuration tab

- 2 To include a section in the report, select the appropriate **Include...** option. The contents of the individual sections are listed in *Table 8*, page 262.

The last section in the report is an index which cannot be excluded from the report.

Project report section (Include option)	Description
Introduction	This section includes user-written text files. See <i>Specifying visualSTATE files to be used as input for Project report</i> , page 263.
Model design	This is the main section of the document. It contains a complete description of the design, including statecharts, transitions, elements, etc.

Table 8: Project report sections

Project report section (Include option)	Description
Model test	This section contains test files, such as Validator static analysis files, Validator dynamic analysis files, Validator test sequence files and Verificator report files.
Model interface	This section contains a table for each element type that is part of the external interface, that is, events, action functions, external variables and constants.
Implementation	This section contains Coder report files.
Pseudo code	This section contains pseudo code for the Project.
Element lists	This section contains a table for each element type, that is, events, event groups, action functions, external variables, internal variables, signals, constants, and external states ^a .

Table 8: Project report sections (Continued)

a. External states are declarations of states defined in another vsr file. The declarations are created automatically by the visualSTATE Designer when states in another vsr file are referenced, for example when using state conditions for a state in another vsr file.

SPECIFYING VISUALSTATE FILES TO BE USED AS INPUT FOR PROJECT REPORT

For specifying the visualSTATE files to use as input for the report, you use the File Input tab of the Documenter Options dialog box (see *General*, page 261).

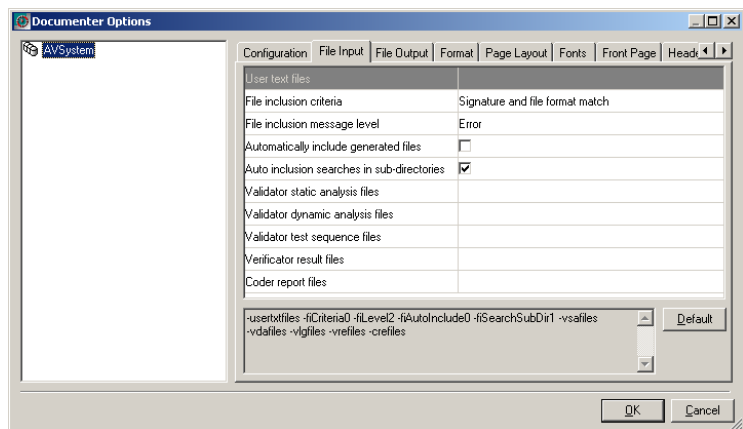


Figure 184: Documenter Options dialog box, File Input tab

The primary input files for the Documenter are the `vsp` and `vsr` files that make up the visualSTATE Project. In addition, you can choose to have the following files included as input for the Project report:

- User text files: Any unformatted text files that you have written. The files are included in the introduction section of the report (see *Table 8*, page 262).
- visualSTATE generated files: Validator static analysis files, Validator dynamic analysis files, Validator test sequence files, Verificator report files and Coder report files.

To ensure consistency between the visualSTATE generated files to be used as input for the report and the visualSTATE Project, the files are checked. By default, visualSTATE generated input files are only included in the report if their digital signatures correspond to the digital signature of the loaded Project.

You can change the level of file check by clicking the ***File inclusion criteria*** option. See *Figure 185*, page 264.

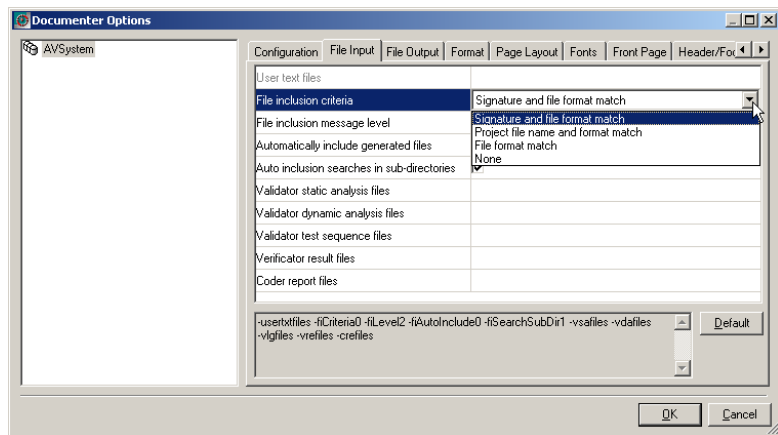


Figure 185: Documenter Options dialog box, file inclusion criteria

To specify type of message to receive on inconsistencies between files selected as input for report, click ***File inclusion message level***.

You can add all visualSTATE generated files by selecting ***Automatically include generated files***. At the time of report generation, the Documenter will search the directory where the `vsp` file is located. To have all subdirectories searched, select ***Auto inclusion searches in subdirectories***.

You can add visualSTATE generated files of a specific type by selecting the file type to add, for example Coder report files, and click the browse button displayed.

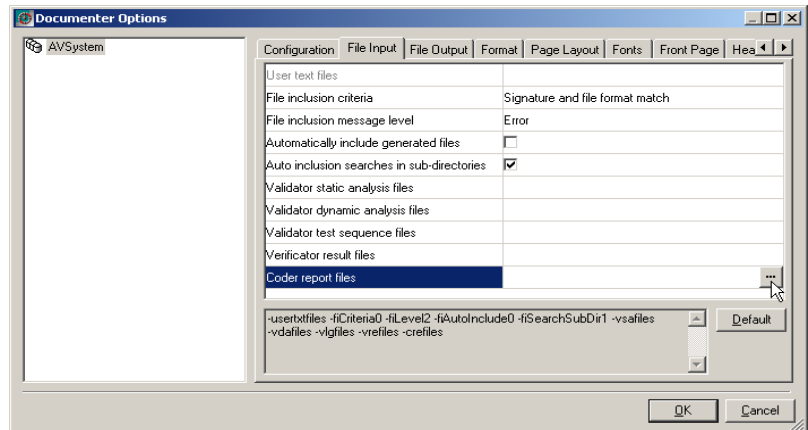


Figure 186: Selecting visualSTATE generated files

In the Select Files dialog box displayed, click the **Auto Add** button to automatically add all files. See Figure 187, page 265.

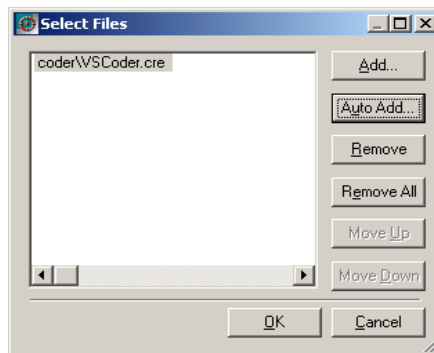


Figure 187: Select Files dialog box

Click the **Add** button to open a dialog box where you can select the file to include. The file is added to the list in the Select Files dialog box. Click **OK**.

Specifying report output format

The output format of the report can be RTF or HTML format. For both output formats, page layouts and fonts can be specified (see *Setting up standard report layout*, page 268).

RTF OUTPUT

The RTF output generated by the Documenter generally conforms to the Rich Text Format (RTF) Specification, version 1.6 (<http://msdn.microsoft.com>), except for the following Documenter-specific RTF fields:

REF:	Used to insert links to bookmarks.
PAGEREF:	Used to insert links to pages.
INCLUDEPICTURE:	Used to insert links to image files (icons and statecharts).
TOC:	Used to insert a table of contents.

Note: The RTF fields may only be understood by MS Word.

Fields of type PAGEREF and TOC are not updated when the RTF output is generated. The easiest way to update the fields is to mark the entire document (press CTRL+A) and then update all fields (press F9).

SPECIFYING RTF OUTPUT FORMAT

- 1 Open the Documenter Options dialog box (as described in *General*, page 261), and click the File Output tab. See *Figure 188*, page 266.

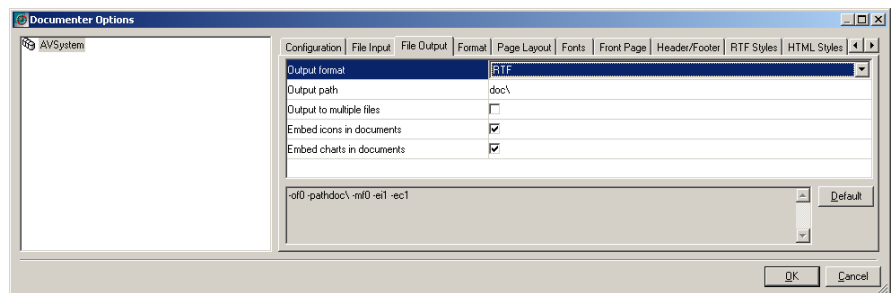


Figure 188: Documenter Options dialog box, File Output tab

- 2 Select **Output format**: RTF.
- 3 To have each report section generated into a separate RTF file, select **Output to multiple files**.

- 4 To embed icons within the generated RTF output, select *Embed icons in documents*.

Note: The generated RTF output may become quite large when embedding icons because the same icon often appears several times in the same document. If this is a problem, deselect *Embed icons in documents*. The icons will be generated into separate files and linked into the generated RTF output.

Note: The option to link icons into the RTF output is non-standard RTF and may only be understood by Microsoft Word.

- 5 To embed statecharts within the generated RTF output, select *Embed statecharts in documents*.

If you deselect *Embed statecharts in documents*, the statecharts will be generated into separate files and linked into the generated RTF output.

Note: The option to link statecharts into the RTF output is non-standard RTF and may only be understood by Microsoft Word.

When you have specified output format, you can set up report layout. See *Setting up standard report layout*, page 268, and *Customizing report layout*, page 271.

HTML OUTPUT

The HTML output generated by the Documenter generally conforms to the HTML 4.01 Specification and the Cascading Style Sheets level 2, CSS2 Specification by W3C (<http://www.w3.org>).

In addition to the HTML report output, a single CSS2 file is generated. The styles of the CSS2 file are based on the options specified on the Page Layout tab, Fonts tab, and HTML Styles tab of the Documenter Options dialog box.

See *Specifying HTML output format*, page 267.

Graphics for reports in HTML format

When you choose to generate a report in HTML format, all images such as icons and statecharts will be generated into separate files that are linked into the HTML output.

Note: Statecharts are generated in EMF format, which is non-standard HTML. Thus statecharts in HTML output may not be visible in all browsers. Microsoft Internet Explorer version 4 or higher can be used to view outputs that include statecharts.

SPECIFYING HTML OUTPUT FORMAT

- 1 Open the Documenter Options dialog box (as described in *General*, page 261), and click the File Output tab. See *Figure 188*, page 266.
- 2 Select *Output format*: HTML.

- 3 To have each report section generated into a separate HTML file, select **Output to multiple files**.

When you have specified output format, you can set up report layout. See *Setting up standard report layout*, page 268, and *Customizing report layout*, page 271.

Setting up standard report layout

This section gives a general description of how to set up page and text layout of the generated report. Advanced layout options are described in *Customizing report layout*, page 271.

The report layout can be customized with regard to

- Page layout. See *Setting up front page layout (RTF output only)*, page 268, and *Setting up page layout*, page 269.
- Text fonts. See *Specifying fonts*, page 270.

SETTING UP FRONT PAGE LAYOUT (RTF OUTPUT ONLY)

- 1 Open the Documenter Options dialog box (as described in *General*, page 261), and click the Front Page tab. See *Figure 189*, page 268.

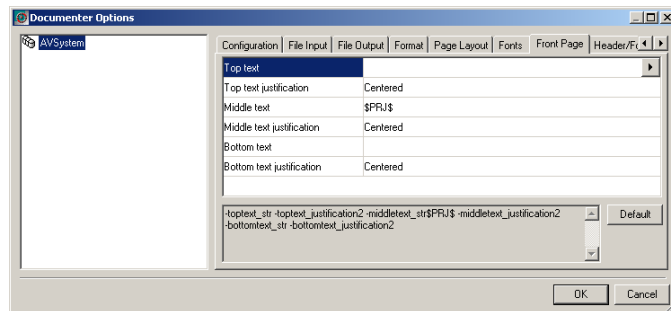


Figure 189: Documenter Options dialog box, Front Page tab

- 2 Here you can specify top text, middle text and bottom text for the front page, along with justification of the texts.

SETTING UP PAGE LAYOUT

- 1 Open the Documenter Options dialog box (as described in *General*, page 261), and click the Page Layout tab. See *Figure 190*, page 269.

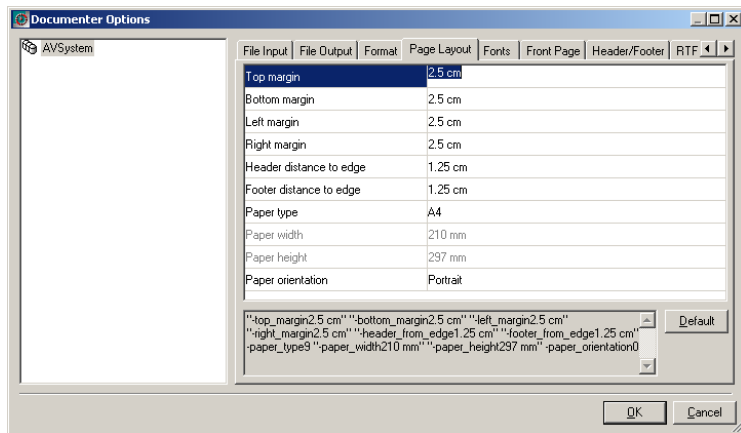


Figure 190: Documenter Options dialog box, Page Layout tab

- 2 Here you can specify margins, paper type and paper orientation.

Length values, such as margins, paper width, paper height, etc., can be entered in different units of measurement (see the online help for available units).

The default values for the page layout options depend on the measurement system specified for your system under Regional Options in the Control Panel.

If the US system is used, all lengths use inches as units and the default paper type is US Letter. If the metric system is used, all lengths use centimeters as units, and the default paper type is A4 Letter.

- To specify headers and footers for the pages after the front page, click the Header/Footer tab, and select the appropriate options. See *Figure 191*, page 270.

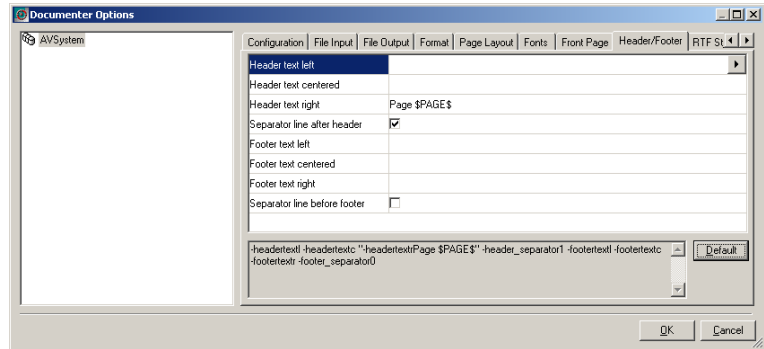


Figure 191: Documenter Options dialog box, Header/Footer tab

Note: The options on the Header/Footer tab can only be set for RTF output.

SPECIFYING FONTS

- Open the Documenter Options dialog box (as described in *General*, page 261), and click the Fonts tab. See *Figure 192*, page 270.

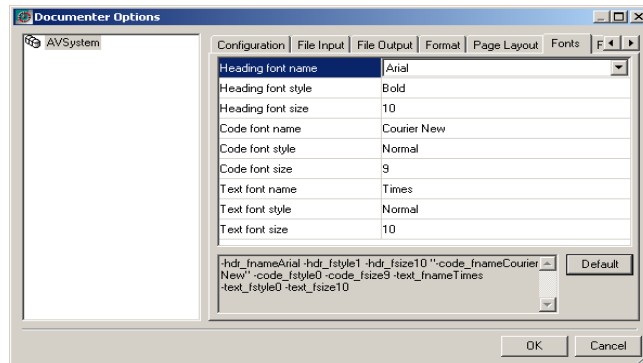


Figure 192: Documenter Options dialog box, Fonts tab

- | | |
|--------------|---|
| Heading font | Used for headings, and text on front page. |
| Code font | Used for code (for example pseudo code and inserted files). |
| Text font | Used for all remaining text. |

Customizing report layout

You can customize the layout for reports in both RTF and HTML format.

CUSTOMIZING LAYOUT FOR REPORTS IN RTF FORMAT

It is possible to use your own styles and templates for a generated report in RTF.

Note: The use of the Documenter RTF style and template options assumes that you are familiar with styles and templates in Microsoft Word.

- 1 Open the Documenter Options dialog box (as described in *General*, page 261), and click the RTF Styles tab. See *Figure 193*, page 271.

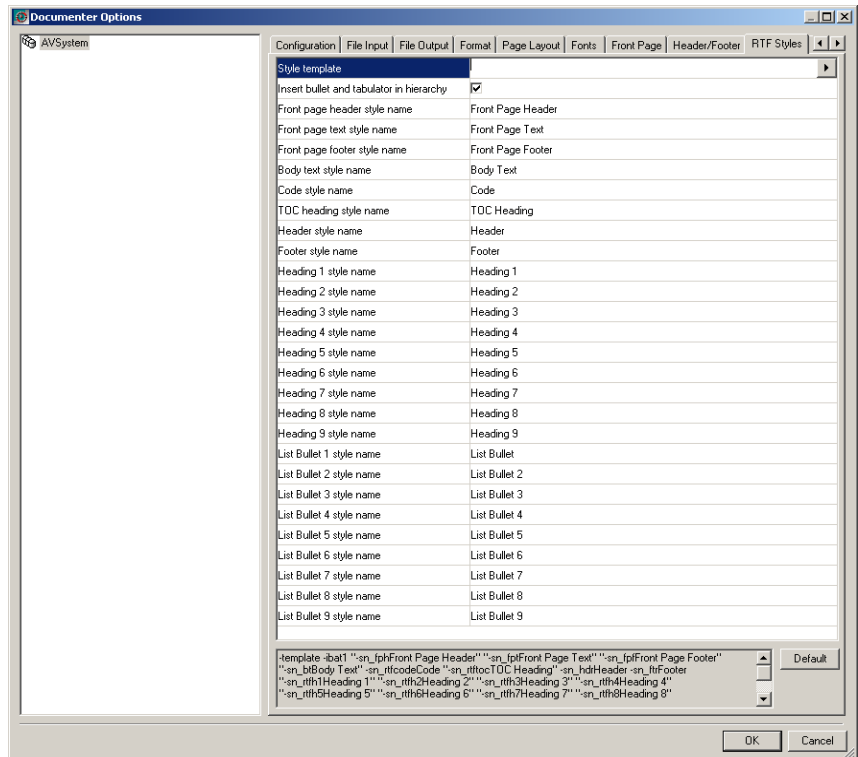


Figure 193: Documenter Options dialog box, RTF Styles tab

- 2 To specify an external template, click *Style template*, and type the path and file name of the template in the field

If Microsoft Word is used for viewing the RTF output generated with an external template, and the style to be applied to the Documenter RTF output is identical to the default style in the default Microsoft Word template `normal.dot`, do the following to have the correct style applied to the generated RTF output:

Modify the RTF style temporarily. For example change the font size for the style, save the template, and change the font size back to its original value.

- 3 To rename a style, click the appropriate style name option, and type the new name in the field.

CUSTOMIZING LAYOUT FOR REPORTS IN HTML FORMAT

You can also use your own styles and style sheets for a generated report in HTML format.

Note: The use of the Documenter HTML style and style sheet options assumes that you are familiar with styles and style sheets in HTML and CSS2.

- 1 Open the Documenter Options dialog box (as described in *General*, page 261), and click the HTML Styles tab. See *Figure 194*, page 272.

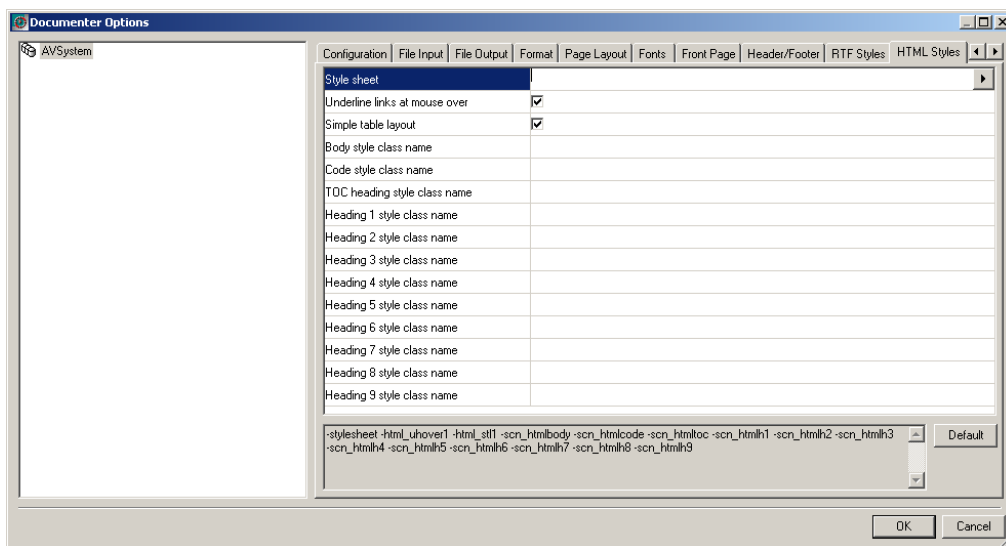


Figure 194: Documenter Options dialog box, HTML Styles tab

- 2 To specify style sheet to link to from the HTML output, select *Style sheet* and type the path and file name of the style sheet in the field.

- 3 If you have chosen to use an existing style sheet, and the sheet contains class names that cannot be changed, you should specify class names for the various HTML elements.

Example

Your existing style sheet file named `company.css` could look as follows:

```
body.company {font-family: "Verdana" serif; font-size: 10pt;
font-style: normal; font-weight: normal;}
```

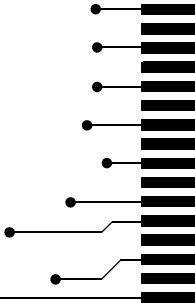
To apply this style sheet to the generated HTML output, do the following:

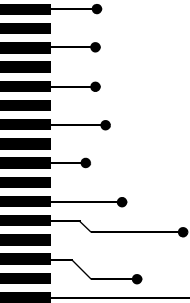
- 1 Click **Heading 1 style class name** and type `company` in the field.
- 2 Click **Style sheet** and type `company.css` in the field.

Part 9: Prototyping

This part of the visualSTATE[®] User Guide includes the following chapters:

- Introduction
- Prototyping with Altia
- Prototype based on visualSTATE generated code
- Prototyping with the visualSTATE Expert DLL.





Introduction

Often you will use visualSTATE to design and generate code for the dynamic behavior or control logic part of an application that has a human/machine interface (HMI).

If you integrate the visualSTATE model with a prototype of your user interface you can combine the test of the human/machine interface with the test of the behavior of the final application at an early stage in your development process. This allows you to continue developing, and refining each part separately.

When designing the control logic part with visualSTATE, you have several options for creating a prototype by means of a graphical model of the user interface and integrate this prototype with your designed visualSTATE model:

- Integration to the visualSTATE model by means of the Altia integration feature of the Validator. This method does not require any programming. See *Prototyping with Altia*, page 279.
- The prototype is created by integrating visualSTATE Coder-generated C code with code developed in a third-party development tool. This approach allows you to use the prototype code directly in the final application. See *Prototype based on visualSTATE generated code*, page 291.
- Prototyping with the visualSTATE Expert DLL. See *Prototyping with the visualSTATE Expert DLL*, page 299.

Prototyping with Altia

When developing your visualSTATE model, you may want to simulate and test a graphical prototype of it. This can be accomplished by using the Altia integration feature of the Validator.

This chapter explains the basic concepts of the Altia integration feature, and how to interface to an Altia design from the Validator. The chapter also describes how to use parameter values when interfacing a visualSTATE model to an Altia design.

For information on how to use the Altia application, refer to the online Altia user documentation.

Note: In this chapter the term *model* will be used to refer to a visualSTATE System. The term *design* will be used to refer to a graphical design created with Altia FacePlate. *Altia* will be used to refer to *Altia FacePlate*.

Basic concepts

By means of the Altia FacePlate application, you can create a graphical prototype of your visualSTATE model. Via the Validator you can connect the visualSTATE model to the Altia design and simulate it.

ALTIA CONNECTION

An *Altia connection* is a communication link between the Validator and an Altia design. Altia designs are created with the Altia application.

When the Validator **Altia Connect** command is activated, the Validator establishes an Altia connection to an Altia design that is automatically loaded into a new instance of the Altia application.

VISUALSTATE ELEMENTS AND ALTIA EXTERNAL SIGNALS

To be able to use the design as a graphical user interface for the model loaded in the Validator, visualSTATE events and action functions must be connected to Altia objects.

If you want a push button in the Altia design to generate a visualSTATE event in the Validator (the same effect as double-clicking an event in the Validator Event window),

you must connect the event to the push button. Likewise you can make a visualSTATE action function turn on an LED object in the design if you connect the action function and the LED object.

Such connections are set up by connecting (or binding) "external signals" defined in the Altia application to graphical objects in the design (see *Connecting visualSTATE elements to Altia objects*, page 283). External signals are either inputs or outputs:

- The external *input* signals are sent from the design to the Validator, that is, they act as events and are often bound to button objects.
- The external *output* signals are sent into the design as actions, for example TURN ON LED2.

The configuration of the external signals is saved along with the graphical layout in the design file (a file with name extension `dsn`).

Figure 195, page 280 shows the Altia main window and the dialog boxes listing external signals and all graphical objects in the loaded `AVSystem` design.

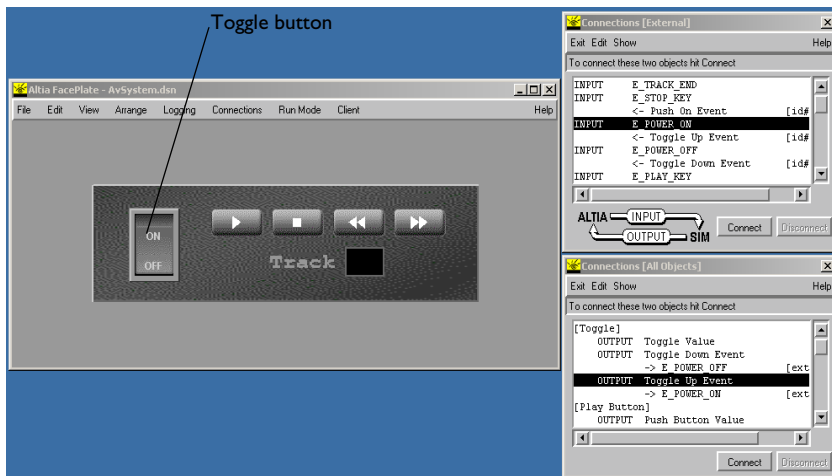


Figure 195: Altia application loaded with the `AVSystem` design

The upper right dialog box named **Connections (External)** shows the list of external signals. For example, the external signal `E_POWER_ON` is connected to the `Toggle Up Event` connector of the Altia object called "Toggle", which is the power button in the design. For more information on external signals, objects and connectors, refer to the Altia user documentation.

The lower right dialog box named **Connections (All Objects)** lists all graphical objects in the design.

When the Validator opens an Altia design, it automatically adds the event names and action function names from the visualSTATE model to the Connections (External) dialog box, unless they are already listed in the dialog box. For example, the dialog box shown in *Figure 195*, page 280 contains the names of the elements in the `AVSystem` model example. The `AVSystem` example has been slightly modified in this chapter, as described in *Using parameters*, page 286.

Existing external signals that already have an event name or action function name are left unchanged by the Validator.

Note that the Validator merely adds external signals to the list, it does not connect them to Altia objects. The external signals and Altia objects must be connected manually as described in *Interfacing a visualSTATE model to an Altia design*, page 281.

Interfacing a visualSTATE model to an Altia design

In order to simulate your visualSTATE model using an Altia design, you must first establish a connection between the two via the Validator.

To interface your visualSTATE model to an existing or new Altia design:

- 1 Start the Validator and load the visualSTATE model to simulate.
- 2 Choose `Altia>Connect` on the menu, or click the **Altia Connect** button (see *Figure 196*, page 282).

An Open Altia Design dialog box will be displayed where you can launch an existing Altia design, or create a new one (see *Figure 197*, page 282).

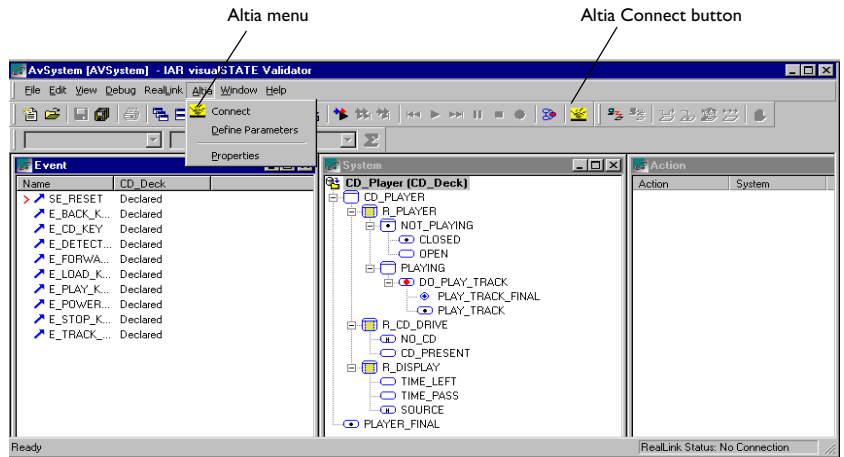


Figure 196: Validator Altia Connect commands

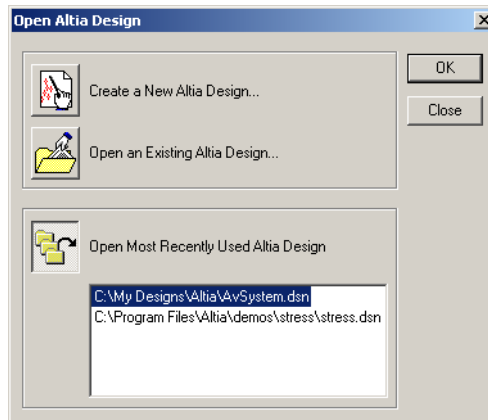


Figure 197: Open Altia Design dialog box (Validator)

- 3 In the Open Altia Design dialog box, select the Altia design to connect to:
 - If the desired design is listed in **Open Most Recently Used Altia Design**, select it from the list. Or

- Click **Open an Existing Altia Design...** button to open a dialog box where you can browse for the desired design file. Click **OK** to load the Altia design into a new instance of the Altia application.

Or create a new design:

- Click **Create a New Altia Design...**, and subsequently click **OK** to open an empty Altia editor. Here you can create the new design right away while the Altia connection is active. For information on how to use the Altia editor, refer to the Altia user documentation.

Whether you connect to an existing Altia design, or create a new one, it is possible to edit it while the Altia connection is active. Any design changes will have immediate effect in the Validator, for example adding new objects and connecting them to the visualSTATE model through new or existing external signal connections (see *Connecting visualSTATE elements to Altia objects*, page 283).

You may even choose to create only the parts of the Altia design that you want to simulate at the moment, and maybe add more objects later.

CONNECTING VISUALSTATE ELEMENTS TO ALTIA OBJECTS

When you have connected your visualSTATE model to an Altia design, the names of the visualSTATE events and action functions will be added to it as new external signals if they are not already contained in it.

The events and action functions that are not connected as external signals to any object in the Altia design are listed in the Validator output window as *unbound visualSTATE events* and *unbound visualSTATE action functions* (see *Figure 198*, page 283).

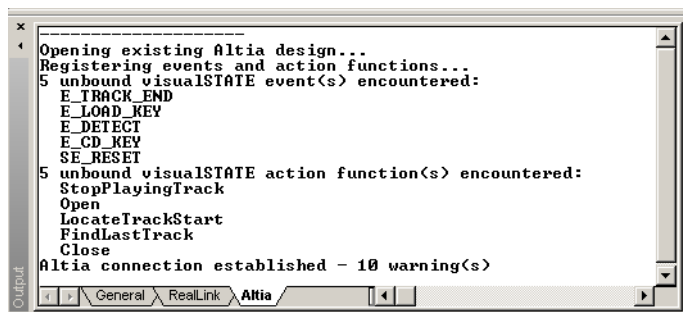


Figure 198: Validator output window, Altia tab

Likewise, all bound external signals in the original Altia design that do not have a visualSTATE event or visualSTATE action function counterpart are listed in the

Validator output window as *unused Altia inputs* and *unused Altia outputs* (see *Figure 198*, page 283).

Note: The lists of unbound visualSTATE events and action functions, and unused Altia inputs and outputs are only written to the Validator output window when Altia connection is established.

- I In Altia, bind the visualSTATE events and action functions you want to use in the Altia design to objects. This binding is done via the Altia Connections (All Objects) and Altia Connections (External) dialog boxes (see *Figure 199*, page 284). You open the dialog boxes by choosing *Connections>All Objects* and *Connections>External Signals* on the Altia menu.

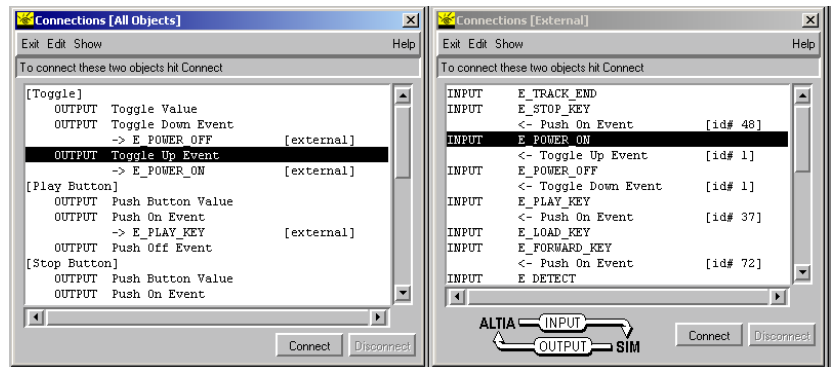


Figure 199: Binding Altia objects to visualSTATE elements

visualSTATE input signals (events) must be bound to Altia *output* objects (marked with **OUTPUT** in the Connections (All Objects) dialog box) and vice versa.

Example of binding a visualSTATE input signal to an Altia output object

- 1 In the Connections (All Objects) dialog box, select the Altia output connector you want to use (marked **OUTPUT**).
- 2 In the Connections (External) dialog box, select the visualSTATE element which you want to bind to the Altia object (marked **INPUT**).
- 3 Click the **Connect** button in one of the dialog boxes. If connection is successful, the status line will read "Connection established", and the Altia object will be bound to the visualSTATE element.

Note that you can bind an external signal to more than one design object, just as you can bind several external signals to the same connector on an object.

To unbind objects and elements connected:

- Select the object/element pair in question, and click the **Disconnect** button in one of the dialog boxes.
- 2 When you have set up the external signal connections you want to, save the Altia design to save the bindings.

Now your visualSTATE model is interfaced with the Altia design, and you can start simulating, as described in *Simulation with Altia*, page 285.

Simulation with Altia

When you have interfaced your visualSTATE model to an Altia design, you can start simulation. You may start simulation even if you have not created a complete Altia design.

- 1 Put the Altia application in *Run mode* by selecting the **Set Run Mode** menu command, or pressing CTRL+D in the Altia application.
- 2 Simulate by sending events and actions between the visualSTATE model and the Altia design.

Events

You can simulate events in two ways: Either by double-clicking the event name in the Event window of the Validator, or by manipulating the corresponding object in the design, provided it is connected to an external input signal

When you send an event into the System using the Validator, the event is also sent to the Altia design where the connected output object is "animated" accordingly, provided the object type supports animation. For example toggle buttons will change from OFF position to ON position.

Action functions

Action functions that are executed in the Validator and connected through an external output signal to an Altia object will have a visible effect in the design, for example the turning on an LED. Note that connected input objects work even if Altia is in *Edit mode*.

Note: Action functions executed in guard expressions and assignments will have no visible effect in the Altia design.

Note: When Altia is in *Edit mode*, it is not possible to manipulate event generators such as buttons in the Altia design, and thus no events will be sent from Altia to the model in the Validator.

Closing the Altia connection

When you are finished using the Altia design, or you want to connect to another design, you close the Altia connection by clicking the Altia toolbar button, or choosing Altia>Disconnect on the Validator menu.

The Altia connection will also be closed automatically when the Validator is closed. Closing the Altia connection does *not* close the Altia application. When you open an Altia connection again, a new Altia instance will be launched.

Using parameters

This section describes why you may need to apply parameters to external Altia connections. The `AVSystem` design included with the visualSTATE software will be used as an example (referred to by the *original AVSystem design*), and it will be explained why it is necessary to modify the original `AVSystem` design when you use parameters for external Altia connections.

In visualSTATE models, events and action functions defined are declared to carry zero or more parameters. For example, in the original `AVSystem` design the `E_POWER_KEY` event has no parameters, while `E_DETECT` is declared with one parameter.

External Altia signals on the other hand always carry one parameter, and many Altia objects accept or emit one parameter. LED objects are input objects that require one parameter for which the values zero and one typically means *turn off* and *turn on* (all parameter values for Altia design objects can be configured). Hence to turn on an LED object, you would typically send an external output signal with the parameter value one.

Toggle buttons are output objects with three connectors (refer to the Altia user documentation). In *Figure 195*, page 280, the `Toggle Value` connector emits a signal with one parameter value for OFF and one for ON. The output object parameters are also used for animation of the objects.

The `E_POWER_KEY` event in the `AVSystem` example is typically connected to a button like the power button in *Figure 195*, page 280. This is a toggle button, which by default uses one as ON value, and zero as OFF value. When you click the power button in the design, an external input signal with the parameter zero or one is sent to the System in the Validator. Because the `E_POWER_KEY` is an event declared without parameters, the parameter from Altia is ignored. Hence clicking the button has the same effect as double-clicking `E_POWER_KEY` in the Validator Event window.

ALTIA PARAMETER VALUES FOR VISUALSTATE EVENTS

So far, it has been quite straightforward to use parameters, but what parameter value should be used for button animation when you click the event in the Validator?

If the button should change from OFF state to ON, the parameter one should be used, and zero should be used in all other cases, provided the default configuration of the button is used. However, it is not possible to express such a relationship in the original AVSystem model.

To achieve correct animation, the E_POWER_KEY event in the original AVSystem design should be replaced by two events: E_POWER_OFF_KEY and E_POWER_ON_KEY. Both events should be connected to the power button shown in *Figure 195*, page 280 through their external input signal connections. The power button is called Toggle in the Connections (All Objects) dialog box (see *Figure 195*, page 280).

The two new events E_POWER_OFF_KEY and E_POWER_ON_KEY must be connected to the Toggle Down Event and Toggle Up Event connectors respectively. These two connectors require different parameter values:

- The Toggle Down Event connector requires the OFF value configured for the toggle button object which by default is zero.
- The Toggle Up Event connector will of course require the ON value.

Consequently the E_POWER_OFF and E_POWER_ON events must be configured to sending the values zero and one respectively as parameters for the external signals. Configuration of events is described in *Assigning Altia parameter values to visualSTATE elements*, page 288.

ALTIA PARAMETER VALUES FOR VISUALSTATE ACTION FUNCTIONS

Like visualSTATE events, visualSTATE action functions must have fixed Altia parameter values assigned to them (this is described in *Assigning Altia parameter values to visualSTATE elements*, page 288).

For example, you could imagine a traffic light system that defines two action functions for each light bulb: GreenOff and GreenOn, etc. These action functions are typically connected through two external output signals to one Altia object for each color. However, this LED object turns on by default when signaled with parameter value one, etc., while the action functions are defined to be parameterless. Thus the action functions must be configured as follows:

- The GreenOff action function should use the parameter value zero for the external connection.
- The GreenOn action function should use parameter value one.

ASSIGNING ALTIA PARAMETER VALUES TO VISUALSTATE ELEMENTS

To assign fixed Altia parameter values to visualSTATE events and action functions, choose Altia>Define Parameters on the Validator menu (see *Figure 196*, page 282). This will open the Define Altia Parameters dialog box shown in *Figure 200*, page 288.

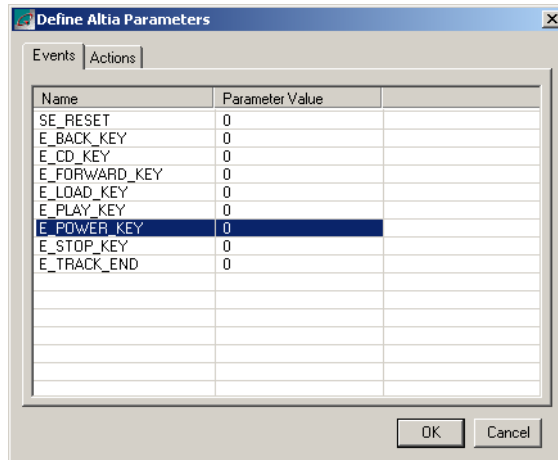


Figure 200: Define Altia Parameters dialog box, Event tab (Validator)

All events and action functions of the visualSTATE model are listed in the dialog box, along with their parameter values which by default are zero.

The parameters are floating point values, but for most Altia objects, such as LEDs and buttons, you typically use integers (0.0 and 1.0). Parameter values in the form of integers are displayed without trailing zero decimals in the Define Altia Parameters dialog box.

Configuring the Altia connection

Typically the default values of the Altia connection will work fine. However, the Altia connection can be configured to suit specific needs.

To configure the Altia connection:

- I Choose Altia>Properties on the Validator menu (see *Figure 196*, page 282). This will open a Define Altia Properties dialog box where you can configure the Altia connection (see *Figure 201*, page 289).

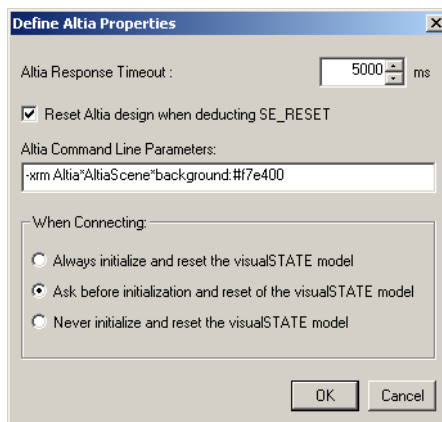


Figure 201: Define Altia Properties dialog box (Validator)

Altia Response Timeout

Here you can specify the number of milliseconds for which the Validator is to wait for a response from the Altia application while establishing an Altia connection.

Reset Altia design when deducting SE_RESET

Select this option to ensure that the Altia design is synchronized with the visualSTATE model upon deduction of the visualSTATE reset event SE_RESET.

Altia Command Line Parameters

Here you can type a series of space-separated arguments to be passed on to the Altia application when it is launched. An example of user-specified command line parameter is shown in *Figure 201*, page 289. For a description of recognized command line parameters, refer to the Altia documentation.

When Connecting

Here you can specify whether the visualSTATE model should be initialized and reset during establishing of the Altia connection.

Note: To ensure synchronization between the visualSTATE model and the Altia design, select the options **Reset Altia design when deducting SE_RESET** and **Always initialize and reset the visualSTATE model**.

-
- 2** When you have configured the Altia connection, click **OK**.

Prototype based on visualSTATE generated code

You can create a software prototype of your visualSTATE model using the visualSTATE Coder-generated C code directly in any third-party development tool that supports ANSI C code.

This chapter gives a general description of how a prototype based on visualSTATE generated code is implemented. It also gives a specific example of how you can integrate the visualSTATE code and visualSTATE APIs with C++ using Microsoft Visual C++.

General

The control logic code generated by the visualSTATE Coder is in C. By means of the visualSTATE APIs, it can be combined with code developed with any third-party development tool that supports the ANSI C standard, for example Microsoft Visual C++, Borland C++ Builder, or Watcom C++.

You implement the prototype application as you would implement a final application (see *Code required for a visualSTATE application*, page 9). This means that you can reuse the control logic designed in visualSTATE from project to project and only write

code for the main loop, and for the handling of events and actions. The principle of this approach is illustrated in *Figure 202*, page 292.

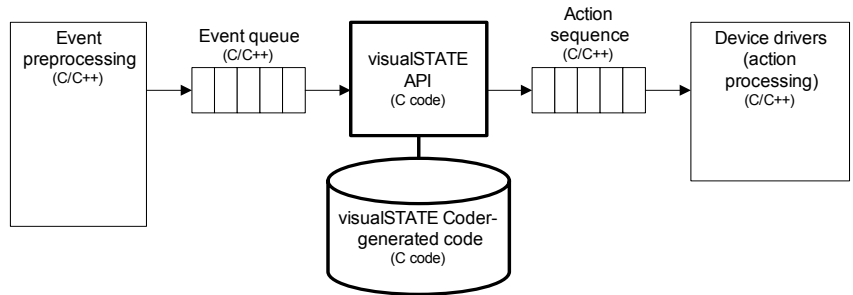


Figure 202: Prototype implementation

Example: Implementing visualSTATE code in C++ code

Creating a prototype in Microsoft Visual C++ differs from creating one in a console application in how the visualSTATE event deduction sequence is implemented. Implementing a while-infinite loop will halt the Windows message loop so this method cannot be used.

Instead you can for example use the following methods:

- Latching onto the Windows idle message by capturing the `WM_IDLEMESSAGE`, for Windows, or `WM_KICKIDLE` message for dialog boxes. Idle messages are sent by Windows when the process has no other messages in the message queue. The frequency of calls to the idle message cannot be determined so an event queue should be implemented for storing and handling visualSTATE events, as described here.
- Using separate threads.

Below is an example of how a prototype can be implemented by capturing the Windows idle message. The example is based on a visualSTATE model with two states: `PowerOn` and `PowerOff`. A statechart of the model is shown in *Figure 203*, page 293.

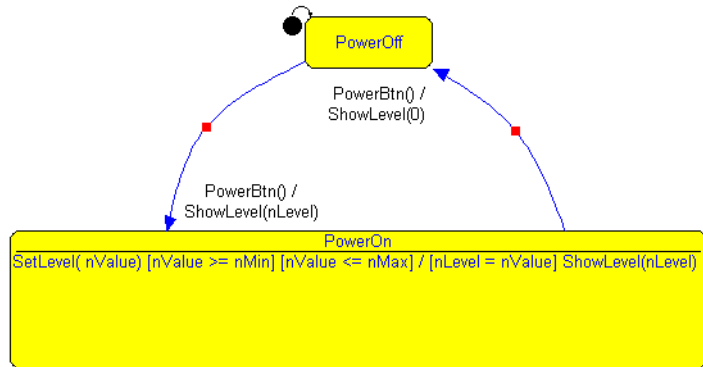


Figure 203: visualSTATE statechart

Switching from state to state is done by triggering the event `PowerBtn`. When the state machine is in the `PowerOn` state, an internal reaction can be triggered by the event `SetLevel`. This internal reaction calls the action `ShowLevel` that can be used to display the event parameter from `SetLevel`.

Implementing the prototype is done in Visual C++ using MFC. The application consists of a dialog box with a button, a slider control and a progress bar (see *Figure 204*, page 293).

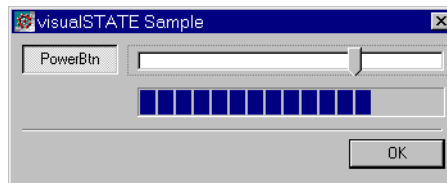


Figure 204: Visual C++ dialog box

The button `PowerBtn` will add the event `PowerBtn` to the event queue. The slider control represents the `SetLevel` event, and the slider position is transmitted as an event parameter. The action `ShowLevel` will activate the progress bar and the action parameter is the display value of the progress bar.

STEPS OF IMPLEMENTATION

- 1 Include the visualSTATE generated code files in your Visual C++ project. Remember to disable the **Precompiled Headers** option for these files, since you are including C files in a C++ project.
- 2 Define an event queue for adding and retrieving events. For an example, see the sample code included with the visualSTATE software.
- 3 Initialize the controls with the constants defined in visualSTATE and initialize the visualSTATE System in the `OnInitDialog` dialog function as shown below.

```

BOOL CVisualSTATESampleDlg::OnInitDialog()
{
    ...
    // nMin and nMax defined in VS as constants

    // Initialize the slider control
    m_hSlider.SetRange(nMin, nMax);
    m_hSlider.SetPos(nMin);

    // Initialize the progress control
    m_hLevel.SetRange(nMin, nMax);
    m_hLevel.SetPos(nMin);

    // Initialize the VS System
    SEM_Init();

    // Initialize the VS System by sending the SE_RESET event
    QueueElement hQe;
    hQe.event      = SE_RESET;
    hQe.parameter = NO_PARAMETER;
    add(hQe);
    ...
}

```

- 4** Map the `PowerBtn` buttons click command to the function `OnPowerBtn`. Map the slider controls slide message by implementing the `OnHScroll` function. The following code shows the message map and the two functions.

```
BEGIN_MESSAGE_MAP(CMainDlg, CDialog)
    ...
    ON_BN_CLICKED(IDC_POWER_BTN, OnPowerBtn)
    ON_WM_HSCROLL()
    ...
END_MESSAGE_MAP()

void CMainDlg::OnPowerBtn()
{
    // add the PowerBtn event onto the queue
    QueueElement qe;
    qe.event      = PowerBtn;
    qe.parameter = -1;
    add(qe);
}

void CMainDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar*
pScrollBar)
{
    CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
    // get slider value and add the SetLevel event onto the
    // queue
    QueueElement qe;
    qe.event      = SetLevel;
    qe.parameter = m_hSlider.GetPos();
    add(qe);
}
```

- 5 Define the implementation of the visualSTATE action ShowLevel as follows:

```
VS_VOID ShowLevel(VS_INT nValue)
{
    // get a handle to the main dialog
    CMainDlg* pDlg = (CMainDlg*)AfxGetMainWnd();
    ASSERT(pDlg);
    // force the dialog to update the progress bar
    pDlg->SetProgressPos(nValue);
}
```

- 6 Implement the visualSTATE event loop by latching onto the Windows message WM_KICKIDLE. The message map and the event loop defined in the OnKickIdle function are shown below.

```
LRESULT CMainDlg::OnKickIdle(WPARAM, LPARAM)
{
    // While events in the event queue
    QueueElement hQe;
    while(retrieve(hQe))
    {
        // Call SEM_Deduct with the event
        unsigned char cc;
        switch(hQe.event) {
            case SE_RESET :
                cc = SEM_Deduct(SE_RESET);
                break;
            case PowerBtn :
                cc = SEM_Deduct(hQe.event);
                break;
            case SetLevel :
                cc = SEM_Deduct(hQe.event, hQe.parameter);
                break;
            default :
                cc = -1; // unknown event
                break;
        }
        if(cc != SES_OKAY)
            ; // Error handler
        // Get resulting action expressions and execute them
        SEM_ACTION_EXPRESSION_TYPE nAction;
        while((cc = SEM_GetOutput(&nAction)) == SES_FOUND)
            SEM_Action(nAction);
        if(cc != SES_OKAY)
            ; // Error handler
        // Change the next state vector
        if((cc = SEM_NextState()) != SES_OKAY)
            ; // Error handler
    }
    return 0L;
}
```

Example: Implementing visualSTATE code in C++ code

Prototyping with the visualSTATE Expert DLL

This chapter describes how to create a visualSTATE prototype using the visualSTATE Expert DLL with Microsoft Visual Basic or C++. It also gives a specific example of how to integrate the visualSTATE Expert DLL with a prototype by means of Microsoft Visual Basic.

For a detailed description of the visualSTATE Expert DLL API functions, see *visualSTATE API Guide*.

What is visualSTATE Expert DLL?

visualSTATE Expert DLL is a binary version of the Expert API delivered as a dynamic link library (DLL). It can be used to interface to a VS Project from a programming language different from C or C++. For example, if it is more convenient to design a graphical user interface (GUI) in Visual Basic, you can write Visual Basic code that interfaces to a VS Project via the Expert DLL.

The principle of this approach is illustrated in *Figure 205*, page 299.

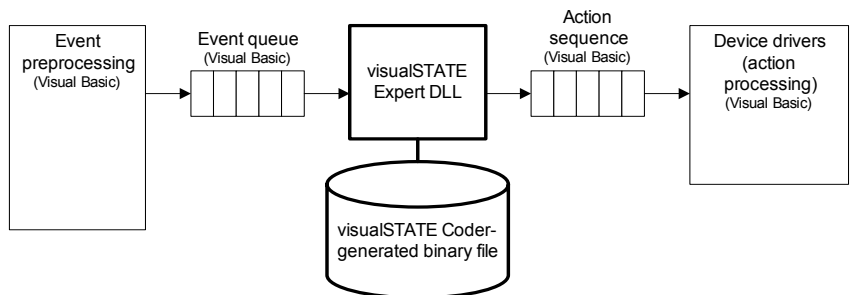


Figure 205: Prototype implementation, visualSTATE Expert DLL

When you use the visualSTATE Expert DLL, the Coder output is a binary file. You specify code generation for the Expert DLL in the Navigator (see *Generating code for the visualSTATE Expert DLL*, page 301).

Note: If the user-written code is written in C or C++, it is recommended not to use the Expert DLL. In this case it is recommended to use the Expert API, and compile the

Expert API source files and user-written source files, and finally link all object files together. See *Prototype based on visualSTATE generated code*, page 291.

EXPERT DLL FILES

The following Expert DLL files are provided:

- The Expert DLL itself, which is named ExpertR9.dll.
- Interface files containing declarations of Expert DLL API functions. The files are used for interfacing to the Expert DLL, and comprise the following:
 - ErrorR9.bas (Visual Basic Expert DLL error file)
 - ExpertR9.bas (Visual Basic Expert DLL interface file)
 - ErrorR9.pas (Borland Delphi Expert DLL error file)
 - ExpertR9.pas (Borland Delphi Expert DLL interface file)
 - ErrorR9.h (C header file containing error definitions)

The Expert DLL files can be used for the programming languages Microsoft Visual Basic 5.0 and Borland Delphi 2.0, and compatible versions of the two languages.

VISUALSTATE PROJECT RESTRICTIONS

When the Expert DLL is used, the following restrictions apply to the VS Project:

- The VS Project may not contain multiple VS Systems.
- The VS Project may not contain events with parameters.
- The VS Project may not contain guard expressions.
- The VS Project may not contain assignments.
- The VS Project may not contain action functions with parameters.
- The VS Project may contain variables, but they are of no use because of the above-mentioned restrictions.

Interaction

Figure 206, page 301 shows how to access the VS Project/VS System via the Expert DLL. The <binary>.sld file is loaded by the Expert DLL and stored in memory.

Accessing the VS Project/VS System is through the API functions of the Expert DLL.

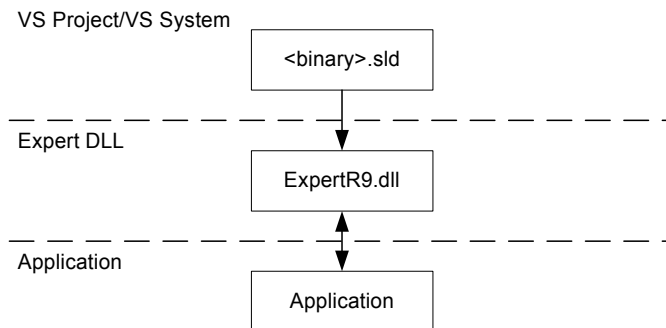


Figure 206: Main flow of information, Expert DLL

Generating code for the visualSTATE Expert DLL

To generate code for the visualSTATE Expert DLL, do the following:

- 1 Launch the Navigator, and open your workspace file.
- 2 Choose Project>Options>Code Generation.... The Coder Options dialog box is displayed. See *Figure 207*, page 301.

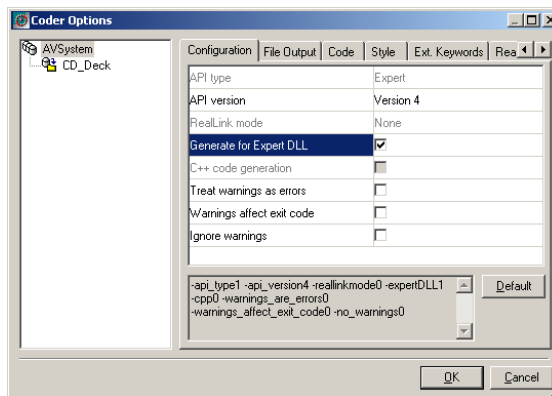


Figure 207: Coder Project Options dialog box, Configuration tab

- 3 In the pane to the left, select the visualSTATE Project containing the System to be code generated for Expert DLL.
- 4 Click the Configuration tab and select **Generate for Expert DLL**. Click **OK**.

- 5 On the menu, choose Project>Code generate.

The visualSTATE System will be code generated into a binary file.

Interfacing to the Expert DLL using Visual Basic

This section demonstrates how to implement a mobile phone in a Visual Basic project using the Expert DLL. The mobile phone example shown in *Figure 208*, page 302 will be used to illustrate the various steps in the implementation process.

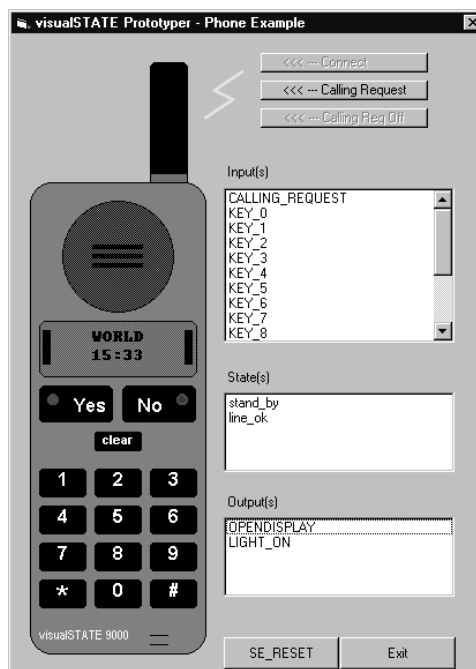


Figure 208: Mobile phone example

The individual steps of the implementation process are described in the following sections:

- *Loading the VS System*, page 303.
- *Loading the VS System and initializing priority queues*, page 303.
- *Activating events*, page 304.
- *Responding to events (event deduction)*, page 305.
- *Listing active events (event inquiry)*, page 306.

- *Retrieving current states*, page 307.
- *Unloading the VS System*, page 308.

The mobile phone example can be converted to other development tools.

For the complete Visual Basic code, see *Appendix C: Source code example*, page 411.

LOADING THE VS SYSTEM

Initially, the VS Project/VS System must be loaded by calling the connecting function SEM_Load. Next, all required initialization functions must be called, in this case only SEM_Init. Notice that the "return value" from SEM_Init must be obtained by a call to SEM_GetInitCC.

Code

```
Private Sub Form_Load()
    Dim cc As Byte

    'load VS System
    cc = SEM_Load("mobile.sld")
    If cc = SES_OKAY Then

        'initialize VS System
        Call SEM_Init

        'get result of initialization
        cc = SEM_GetInitCC()
        If cc <> SES_OKAY Then
            Call SEM_VBErrorHandler("SEM_InitCC", cc)
        End If
    Else
        Call SEM_VBErrorHandler("SEM_Load", cc)
        MsgBox "Program terminated..."
    End
    End If
End Sub
```

LOADING THE VS SYSTEM AND INITIALIZING PRIORITY QUEUES

Events are placed in queues prior to a deduction. A maximum of 10 queues can be defined, with priorities ranging from 1 to 10 and 10 having the highest priority. Whenever a deduction is performed, the next event is the first inserted event in the high-priority queue.

The code below is an alternative to the code shown in *Loading the VS System*, page 303.

Code

```

Private Sub Form_Load()
    Dim cc As Byte

    'load VS System
    cc = SEM_Load("mobile.sld")
    If cc = SES_OKAY Then

        'Initialize VS System
        Call SEM_Init

        'get result of initialization
        cc = SEM_GetInitCC()
        If cc <> SES_OKAY Then
            Call SEM_VBErrorHandler("SEM_InitCC", cc)
        End If

        'initialize queues
        Call SEM_QueueInit

        'define queue 1 with low priority and room for 4 inputs
        cc = SEM_QueueCreate(1, 1, 4)
        If cc <> SES_QUEUE_OKAY Then
            Call SEM_VBErrorHandler("SEM_QueueCreate", cc)
        End If

        'define queue 2 with high priority and room for 4 inputs
        cc = SEM_QueueCreate(2, 10, 4)
        If cc <> SES_QUEUE_OKAY Then
            Call SEM_VBErrorHandler("SEM_QueueCreate", cc)
        End If
    Else
        Call SEM_VBErrorHandler("SEM_Load", cc)
        MsgBox "Program terminated..."
    End If
End Sub

```

ACTIVATING EVENTS

The mobile phone contains two types of events: events activated from buttons, and other events. Other events are events that have no buttons associated with them.

'CALLING_REQUEST' is one such event. It is sent from the operator to the mobile phone when a call is being received.

Both types of events are handled in the same way and can be activated from the 'Input(s)' list box.

Code for the button 'NO'

```

Private Sub But_No_Click()
    Dim cc As Byte

    'insert event into queue one
    cc = SEM_QueuePut(1, "KEY_NO")
    If cc = SES_QUEUE_OKAY Then
        'call function for event, state and output handling
        Call DispatchOutput
    Else
        Call SEM_VBErrorHandling("SEM_QueuePut", cc)
    End If
End Sub

```

'KEY_NO' is the event identifier name defined in the <hdata>.h file.

RESPONDING TO EVENTS (EVENT DEDUCTION)

The code shown in *Activating events*, page 304, adds the event to the queue and then calls the function 'DispatchOutput'. 'DispatchOutput' empties the queue(s), taking one event at a time and calling SEM_Deduct, thereby triggering a transition.

After calling SEM_Deduct, retrieve action expressions by calling SEM_GetOutput.

Code for 'DispatchOutput'

```

Public Sub DispatchOutput()
    Dim cc As Byte
    Dim cc1 As Byte
    Dim event As Integer
    Dim iptr As Integer
    Dim str As String * 129
    Dim trimstr As String
    Static Busy As Boolean

    If Busy <> True Then
        Busy = True

        'while still events in queue(s)
        Do While SEM_QueueAllGet(event) = SES_QUEUE_OKAY
            'fire event
            cc = SEM_Deduct(event)
            If cc <> SES_OKAY Then
                Call SEM_VBErrorHandler("SEM_Deduct", cc)
            End If

            Do 'while output found

```

```

'get next output number
cc = SEM_GetOutput(iptr)
If cc = SES_FOUND Then

    'get output name from output number
    If SEM_Name(OUTPUTTYPE, iptr, str, 128) =
        SES_OKAY Then
        'convert output string to Visual Basic string
        Call RemoveAsciiZeroAndTrim(str, trimstr)

        'Activate output drivers
        Select Case trimstr
            Case "CLEAR_DISP"
                (output driver for 'CLEAR_DISP')
                ...
                ...
            Case "UPDATE_DISP"
                (output driver for 'UPDATE_DISP')
            Case Else
                MsgBox "Output Var." & "'" & trimstr & "'" &
                    "is not Defined"
        End Select
    End If
End If
'continue until no more outputs
Loop Until cc <> SES_FOUND
If cc <> SES_OKAY Then
    Call SEM_VBErrorHandler("SEM_GetOutput", cc)
End If

'function for retrieving current states
Call GetCurrentStates

'function for retriving active events
Call GetActiveEvents

'continue until queues are empty
Loop
Busy = False
End If
End Sub

```

LISTING ACTIVE EVENTS (EVENT INQUIRY)

After each call to SEM_Deduct, the VS System changes state(s), and new events become active. Calling the function GetActiveEvents can retrieve the active events.

Code for the function GetActiveEvents

```

Public Sub GetActiveEvents()
    Dim cc As Byte
    Dim iptr As Integer
    Dim str as String * 129
    Dim trimstr As String
    cc = SEM_Inquiry
    If cc <> SES_OKAY Then
        Call SEM_VBErrorHandler("SEM_GetInput", cc)
    End If

    Do 'while still active events

        'get active event number
        cc = SEM_GetInput(iptr, 0)

        If cc = SES_FOUND Then

            'get event name from event number
            If SEM_Name(EVENT_TYPE, iptr, str, 128) = SES_OKAY Then

                'convert string to Visual Basic string
                Call RemoveAsciiZeroAndTrim(str, trimstr)

                'activate active event drivers
                Select Case trimstr
                    Case "CALLING_REQUEST"
                        (active event handling for 'CALLING_REQUEST')
                    ...
                    Case "WEAK_SIG"
                        (active event handling for 'WEAK_SIG')
                End Select
            End If
        End If

        'until no more active events
        Loop Until cc <> SES_FOUND

        If cc <> SES_OKAY Then
            Call SEM_VBErrorHandler("SEM_GetInput", cc)
        End If
    End Sub

```

RETRIEVING CURRENT STATES

After each call to SEM_Deduct, new current states are found. Calling the function GetCurrentStates can retrieve these current states.

Code for the function GetCurrentStates

```

Public Sub GetCurrentStates()
    Dim cc As Byte
    Dim is_on As Byte
    Dim iState As Integer
    Dim str As String * 129
    Dim trimstr As String
    Dim Machine As Integer

    'loop through all state machines
    For Machine = 0 To SEM_NoMachines - 1 Step 1

        'get current state in state machine
        cc = SEM_State(Machine, iState)
        If cc <> SES_FOUND Then
            Call SEM_VBErrorHandler("SEM_State", cc)
            Exit For
        End If

        'get state name from state number
        cc = SEM_Name(STATE_TYPE, iState, str, 128)
        If cc <> SES_OKAY Then
            Call SEM_VBErrorHandler("SEM_Name", cc)
            Exit For
        End If

        'convert string to Visual Basic string
        Call RemoveAsciiZeroAndTrim(str, trimstr)

        'Activate current state driver
        Select Case trimstr
            Case "call_wait"
                (handling for current state 'call_wait')
            ...
            Case "tryconnect"
                (handling for current state 'tryconnect')
        End Select

        'get next state machine
    Next Machine

End Sub

```

UNLOADING THE VS SYSTEM

When terminating the application, the VS System must be unloaded and queues must be removed from memory.

Code for the Terminate function

```
Private Sub Form_Terminate()  
    Call SEM_Free  
    SEM_QueueDestroy(1)  
    SEM_QueueDestroy(2)  
End Sub
```


Part 10: Working in an OSEK environment

This part of the visualSTATE[®] User Guide includes the following chapters:

- Using the visualSTATE OSEK Kit
- Building a runtime application
- Runtime considerations.





Using the visualSTATE OSEK Kit

Runtime applications developed with visualSTATE can be used with or without a real-time operating system. If you choose to use OSEK as operating system, you can use the visualSTATE OSEK Kit which provides a user-friendly interface to using the visualSTATE software in an OSEK environment.

The visualSTATE OSEK Kit is launched via the Navigator and consists of a visualSTATE OSEK API and a wizard.

This chapter describes how to enable OSEK support in visualSTATE and assign visualSTATE Systems to OSEK tasks.

Generating visualSTATE files for use in an OSEK environment

To be able to generate visualSTATE files for use in the OSEK environment, the following steps are necessary:

- 1 OSEK support must be enabled in the Navigator, and an OIL file must be selected (see *Enabling OSEK support*, page 313).
- 2 A complete OIL file must be generated by means of an OIL file builder, for example Motorola's OSEK Builder. From this OIL file, the visualSTATE OSEK wizard extracts information about tasks, messages etc. See your OSEK file builder documentation.
- 3 The visualSTATE OSEK wizard must be run (see *Assigning visualSTATE Systems to OSEK tasks*, page 315).

Enabling OSEK support

OSEK support in visualSTATE can be enabled for one or more visualSTATE Projects via the Navigator, as follows:

- I Start the Navigator, and open the workspace that contains the visualSTATE Project(s) for which to enable OSEK support.

- 2 In the workspace browser, select the appropriate Project, open the pop-up menu and choose Options>OSEK. An OSEK Options dialog box is displayed. See *Figure 209*, page 314.

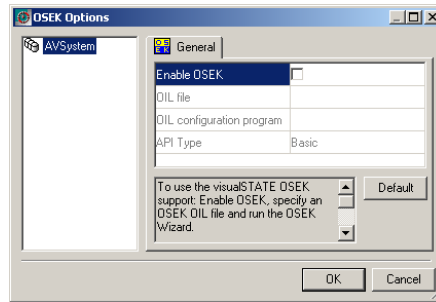


Figure 209: Navigator Settings dialog box, OSEK page

- 3 Select the **Enable OSEK** check box to enable OSEK support for the Project. The other options of the dialog box will become available. See *Figure 210*, page 314.

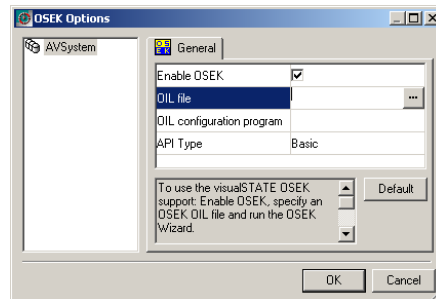


Figure 210: OSEK support enabled

- 4 Click **OIL File** and click the browse button to find the OIL file to use for the visualSTATE System.
- 5 Click **OIL Configuration Program** and click the browse button to specify the OIL configuration program to use for the visualSTATE Project. The program is used for editing the OIL file. If no program is specified here, your system's default OIL configuration program will be used for editing the OIL files used in the visualSTATE Project.

Now the OSEK OIL file has been enabled which is indicated by an icon in the Navigator workspace browser.

You can edit the OIL file by double-clicking the OSEK icon in the workspace browser. This will launch the OIL configuration program that you have specified. If no program was specified, the default OIL configuration program will be launched.

Assigning visualSTATE Systems to OSEK tasks

By means of the visualSTATE OSEK wizard it is possible to specify which visualSTATE Systems are to be run in which OSEK tasks. This information is stored in two ANSI C files that are generated by the visualSTATE OSEK wizard.

Note: Before running the visualSTATE OSEK wizard, the complete OSEK OIL file to be used for the visualSTATE Project should be built in the OSEK OIL file builder.

To run the visualSTATE OSEK wizard:

- 1 Launch the Navigator and open your workspace. Ensure that the correct Project has been set as active.
- 2 Choose Tools>OSEK Wizard.

The visualSTATE OSEK wizard is started which will load the specified OIL file. See *Figure 211*, page 315.

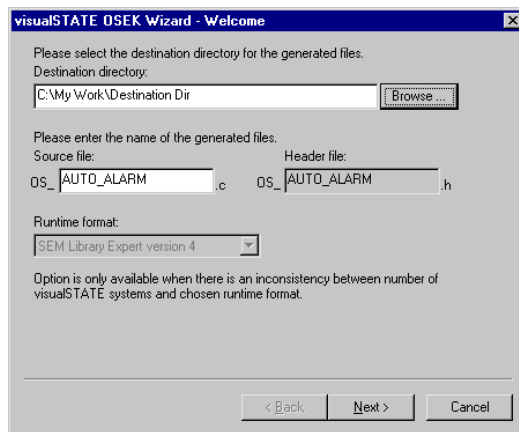


Figure 211: OSEK wizard, first page

- 3 Specify destination directory. This is the directory where the wizard-generated ANSI C files are to be stored. Default destination directory is the same directory as the one specified for the visualSTATE Project file (file name extension `vsp`).

4 Specify source and header file names of the ANSI C files that will be generated by the visualSTATE OSEK wizard. If existing file names are specified, you have the option of overwriting the existing file name, appending to the file name, or canceling the operation. The names of the two files generated by the wizard are automatically prefixed with `OS_` to avoid name conflicts in the final runtime application.

5 The Runtime format list shows the current OSEK options. If the visualSTATE Project only contains one visualSTATE System, and the value for the OSEK option *API type* is *Expert* (see *Figure 210*, page 314), a warning is given when you proceed to the next wizard page.

Note: The runtime footprint of the visualSTATE Basic API is smaller than that of the Expert API. If there is only one visualSTATE System in the visualSTATE Project, use the Basic API.

OSEK options are specified in the Navigator as described in *Enabling OSEK support*, page 313.

6 Click *Next* to proceed to the next wizard page. See *Figure 212*, page 316.

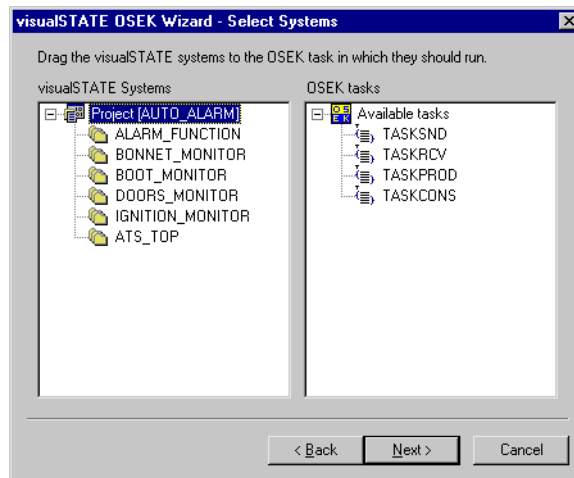


Figure 212: OSEK wizard, Select Systems

7 Drag the visualSTATE Systems into the OSEK tasks in which they should run. It is possible to assign multiple visualSTATE Systems to one OSEK task.

Example

See *Figure 213*, page 317. The visualSTATE System `ALARM_FUNCTION` has been assigned to the OSEK task `TASKSND`. The visualSTATE Systems in the left pane of the window have not been assigned to any OSEK tasks yet.

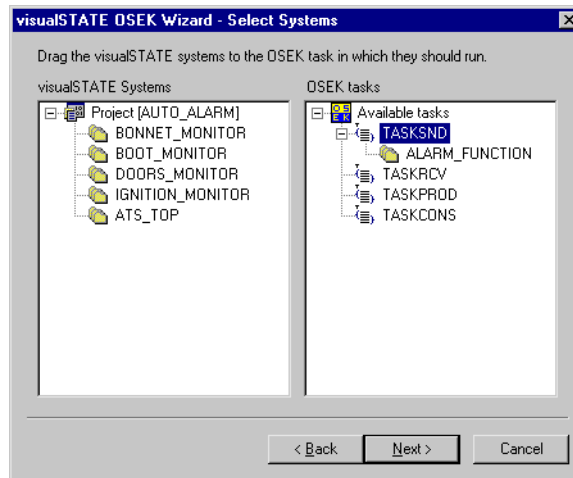


Figure 213: visualSTATE System assigned to an OSEK task

- 8 When you have assigned visualSTATE Systems to the appropriate OSEK tasks, click *Next* to proceed to the next wizard page. See *Figure 214*, page 318.

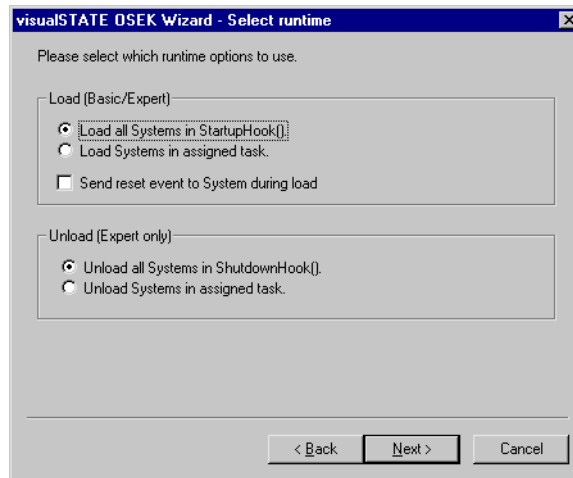


Figure 214: OSEK wizard, Select runtime options

- 9 Because each visualSTATE System must be loaded before use, you must specify load options:
- Select **Load all Systems in StartupHook()** to have the visualSTATE Systems loaded during OSEK startup.
 - Select **Load Systems in assigned task** to have the visualSTATE Systems loaded during OSEK task activation.
- 10 In the runtime application, the visualSTATE reset event `SE_RESET` must always be sent into the visualSTATE System before any other events can be sent into it. Select **Send reset event to System during load** for automatic reset of each visualSTATE System during load.

Note: If the **Send reset event to System during load** check box is cleared, you must incorporate the sending of the event in the runtime application.

The **Unload** option only applies when the Expert API is used. Each visualSTATE System allocates a block of memory that must be released at some time. These memory blocks can be released either when the specified OSEK task is closed, or when OSEK shuts down.

- 11 Click *Next*. The Summary page of the wizard will be displayed (see example in *Figure 215*, page 319).

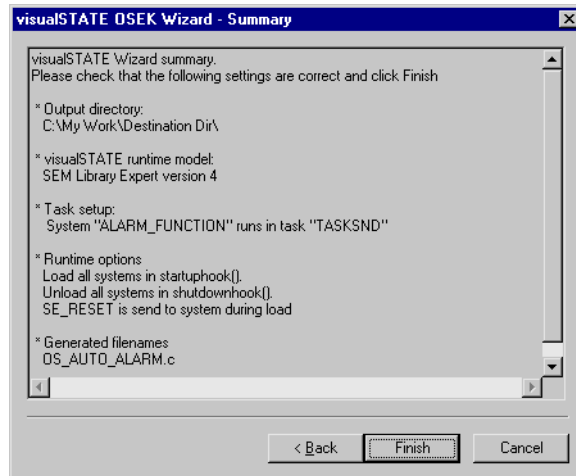


Figure 215: OSEK wizard, Summary

The Summary page shows the options specified. Check that they are correct before clicking **Finish**.

Building a runtime application

This chapter describes how to build a runtime application with the ANSI C files generated by the visualSTATE OSEK wizard.

Requirements for building a runtime application

The following is required for building a runtime application with the ANSI C files generated by the visualSTATE OSEK wizard:

- OSEK API files (supplied by the OSEK vendor).
- The two ANSI C files generated by the visualSTATE OSEK wizard (prefixed `OS_`). The files are found in the destination directory you specified on the first page of the visualSTATE OSEK wizard (see *Figure 211*, page 315).
- Two visualSTATE OSEK API files named `VS_OSEK.C` and `VS_OSEK.H`, which are used by the wizard-generated ANSI C files in the runtime application. These two API files are a thin wrapper around the standard visualSTATE API functions with some additional information.
- Additional source code that is necessary for supplying visualSTATE with events. This source code must be provided by the developer. In addition, the developer must write the functions for each of the actions defined in visualSTATE.

Figure 216, page 322 shows the components required for building a runtime application in visualSTATE.

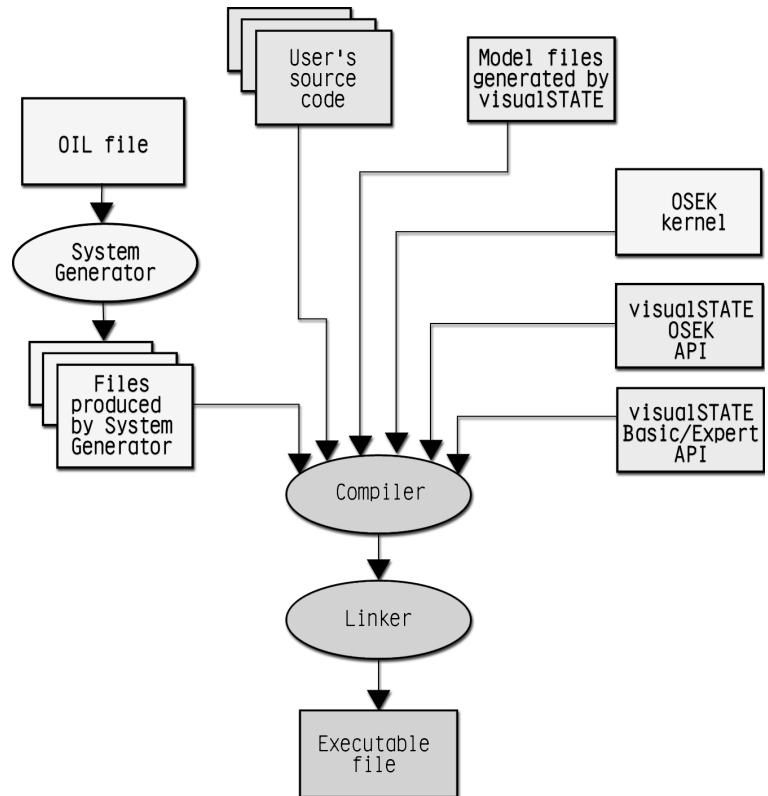


Figure 216: Components required for a runtime application

Exported visualSTATE OSEK API functions

The following visualSTATE OSEK API functions are exported and used in the code generated by the visualSTATE OSEK wizard:

<code>visualSTATE_Load()</code>	Loads and initializes a visualSTATE System. When the Basic API is used, the function does not take any parameters whereas the Expert API takes a context pointer and a system ID.
<code>visualSTATE_Deduct()</code>	Performs a deduction on the supplied event. Actions and assignments are executed, and visualSTATE changes to the next state. The Basic API takes one event as parameter, whereas the Expert API also requires a system ID.
<code>visualSTATE_Unload()</code>	Unloads the visualSTATE Systems from memory. Only available in Expert API where a system ID is required as parameter.

Table 9: Exported OSEK API functions

Note: To select Expert API for the runtime application, define `VS_EXPERT_MODE` in the compiler preprocessor. If `VS_EXPERT_MODE` is not defined, the Basic API will be applied by default.

Supplying events

A typical loop in visualSTATE has the following structure (in the following example some lines and syntax have been removed for clarity):

Example

```
while(TRUE)
{
    /* Wait for OS signal */
    WaitEvent(USER_EVENT);

    /* Get event */
    EventNo = EventHandling();

    /* Process the event in visualSTATE */
    visualSTATE_Deduct(vs_context_pointer, EventNo);
}
```

Because no task may consume 100% of the CPU time, there must be some sort of scheduling. Inside the loop, you must add code that waits for OSEK to signal the task to start.

visualSTATE must be supplied with events from an external source. The event can be anything from physical events in the external hardware environment to events that occur in other parts of the software.

Since OSEK supports messages being sent from one task to another, one possible solution is to have a single task that handles all external events and via messages sends them to the OSEK task(s) where visualSTATE runs.

Example

```
while(TRUE)
{
    /* Get message */
    ReceiveMessage( MyMsg, _MyMsg );

    /* Process the event in visualSTATE */
    visualSTATE_Deduct(vs_context_pointer, _MyMsg.EventNo);
}
```

API examples

This section gives two examples of how to use the code generated by the visualSTATE OSEK wizard in a runtime application. The first example describes code generated for the Basic API, and the second example describes code generated for the Expert API.

For a detailed description of the visualSTATE standard APIs, refer to *visualSTATE API Guide*

BASIC API

Before any events can be processed, the entire visualSTATE System must be loaded and initialized in the runtime application. This is done by calling the function `visualSTATE_Load()`. The function must only be called once during the lifetime of the runtime application, and it is therefore well-suited for `StartupHook()` or `main()`.

Example

```
int main(void)
{
    /* load and initialize visualSTATE Basic */
    visualSTATE_Load();

    /* Start OSEK */
    StartOS( Mode );
}
```



```

    return 0;
}

```

When `visualSTATE_Load()` has been called, `visualSTATE` is ready for normal operation.

Events can be processed by using `visualSTATE_Deduct()` which always returns a completion code. In case of an error, you should take the necessary action. Completion codes are found in the Basic API file `SEMLibB.h`.

If event parameters are used, the parameters for `visualSTATE_Deduct()` change slightly because `visualSTATE_Deduct()` is defined as a macro. The event and all its parameters must therefore be enclosed in a pair of parentheses, as shown in the following example.

Example

```

unsigned char nRes;
nRes = visualSTATE_Deduct((EventNo, param1, ..., paramN));
if(nRes != SES_OKAY)
{
    /* An error occurred */
}

```

EXPERT API

If there are multiple `visualSTATE` Systems, the Expert API is used. Each `visualSTATE` System must be loaded and initialized in the runtime application before use.

Load and initialization of the `visualSTATE` System are by means of the `visualSTATE_Load()` function which takes the following parameters:

- A pointer to a `visualSTATE` context pointer
- A unique system identifier.

Since each `visualSTATE` System can be configured independently of the others, the wizard generates a static structure that contains the setup for each `visualSTATE` System. In addition to this structure, there is an array of `visualSTATE` context pointers.

Each `visualSTATE` System is referred to via a specific context pointer that must be supplied when using the `visualSTATE_Deduct()` and `visualSTATE_Unload()` functions. The context pointer is assigned during the calling of the `visualSTATE_Load()` function.

Loading of a visualSTATE System is either via the `StatupHook()`, `main()`, or during OSEK task activation, as specified in the visualSTATE OSEK wizard (see *Figure 214*, page 318). An example of load during task activation is shown in the following:

Example

```
TASK (MyTask)
{
    CONTEXT* vs_context_pointer;
    unsigned char nRes;

    /* load and initialize expert system 1 */
    visualSTATE_Load(&vs_context_pointer, VS_SYSTEM_1);

    while(TRUE)
    {
        /* Wait for OS event */
        WaitEvent(MyEvent);

        /* Get event */
        EventNo = EventHandling();

        nRes = visualSTATE_Deduct(vs_context_pointer, EventNo);
        if(nRes != SES_OKAY)
            break;
    }

    visualSTATE_Unload(vs_context_pointer);
    TerminateTask();
}
```

In this example, first a context pointer is created. While system 1 is loaded, it will be assigned to the visualSTATE System. When a visualSTATE deduct function is to be run, the context pointer together with the event is supplied. In case the deduct should fail, the `while(TRUE)` loop is broken, the visualSTATE System is unloaded, and the task terminated.

Note: `visualSTATE_Load()` and `visualSTATE_Unload()` must always be used in pairs. If the same visualSTATE System is loaded twice without being unloaded in between, a memory leak will occur.

If event parameters are used, the parameters for `visualSTATE_Deduct()` change slightly because `visualSTATE_Deduct()` is defined as a macro. The event and all its parameters must be enclosed in a pair of parentheses, as shown in the following example.

Example

```

unsigned char nRes;
nRes = visualSTATE_Deduct(vs_context_pointer, (EventNo, param1,
..., paramN));

if(nRes != SES_OKAY)
{
    /* An error occurred */
}

```

If there are global external variables that must be initialized, the following code must be inserted into the `main()` function, as follows:

```

int main(void)
{
#ifdef PROJECT_INIT_EXTERNAL_VARIABLES_NAME
    (*PROJECT_INIT_EXTERNAL_VARIABLES_NAME)();
#endif
    StartOS( Mode );
    return 0;
}

```

Global external variables should not be initialized more than once, so the `main()` function is a most suitable place to put them.

Runtime considerations

This chapter describes the stack usage and RAM/ROM usage by the visualSTATE APIs when using the visualSTATE OSEK API.

Stack usage

It is possible to determine stack usage when using the visualSTATE OSEK API. See *Calculating stack usage*, page 329.

The stack usage values listed *Table 15*, page 331 and *Table 16*, page 332 may not be the maximum values. Stack usage depends on whether event parameters and action parameters are used. For example, if event parameters are used, stack usage for the `visualSTATE_Deduct` function increases by the variables used. An event that takes two parameters of the type integer will cause stack usage to increase by 4 bytes. The use of guard conditions, actions (without parameters) and assignments will not increase stack usage.

Stack usage by the Basic API and Expert API functions are listed in *Table 10*, page 329 and *Table 11*, page 329 respectively. In the calculation of stack usage, the same assumptions as those applied in *Table 14*, page 331 were used.

visualSTATE OSEK API function	Stack usage in bytes
<code>visualSTATE_Load</code>	1
<code>visualSTATE_Deduct</code>	3

Table 10: Stack usage by Basic API

visualSTATE OSEK API function	Stack usage in bytes
<code>visualSTATE_Load</code>	20
<code>visualSTATE_Deduct</code>	18
<code>visualSTATE_Unload</code>	3

Table 11: Stack usage by Expert API

CALCULATING STACK USAGE

In order to calculate the exact stack usage some runtime information must be known. A small runtime application with less than 256 events, 256 states, 256 actions etc. will result in the smallest possible stack usage. However a full-blown real-world application may use more than 256 states.

The sizes of the following types are determined by the size of the runtime application:

SEM_EVENT_TYPE	The number of events in the project. A size of more than 1 byte is unlikely in many applications as it would require more than 256 events.
SEM_INSTANCE_TYPE	The number of instances of a given System. Most likely to have a size of 1 byte.
SEM_ACTION_EXPRESSION_TYPE	Size is determined by the total number of actions and assignments. Likely to have a size of 2 bytes.
SEM_EXPLANATION_TYPE	Size is determined by the total number of explanations. Only available if you specified this option in the Coder. Assumed to be 0 bytes (not used).
SEM_STATE_MACHINE_TYPE	Size is determined by the total number of state machines. Unlikely to have a size of more than 1 byte.
SEM_STATE_TYPE	Size is determined by the total number of states in a given System. Likely to have a size of 2 bytes.

Table 12: Type sizes determined by runtime application size

The sizes of the following types are determined by the compiler, linker and target hardware:

pointer	The size of a pointer. On small 8-bit processors it is likely to have a size of 2 bytes, whereas large 32-bit processors use 4 bytes to store a pointer. Some may even store a pointer in a CPU register thus requiring no stack space.
short	The size of variable of type short is larger than the size of a byte and smaller or equal to the size of an integer. Likely to have the same size as an integer.
int	The size of an integer is typically 2 bytes for a small 8-bit processor, whereas a large 32-bit processor requires 4 bytes.
long	The size of a variable of type long is most likely to be 4 bytes.

Table 13: Type sizes determined by compiler, linker and target hardware

Stack sizes

Table 15, page 331 and *Table 16*, page 332 list stack usage by the Basic API and Expert API. The calculated stack sizes in the tables are based on the assumptions in *Table 14*, page 331.

Because stack usage depends on whether event parameters and action parameters are used, the stack sizes listed *Table 15*, page 331 and *Table 16*, page 332 should only be

regarded typical sizes that apply to the assumptions in *Table 14*, page 331. In the runtime application, stack sizes will depend on the actual hardware and software platform used.

visualSTATE type	Size in bytes
SEM_EVENT_TYPE	1
SEM_INSTANCE_TYPE	1
SEM_ACTION_EXPRESSION_TYPE	2
SEM_EXPLANATION_TYPE	0
SEM_STATE_MACHINE_TYPE	1
SEM_STATE_TYPE	2
Pointer	2
Short	2
int	2
long	4

Table 14: Assumptions for stack size calculation

The return address of the visualSTATE API function is not included in the stack sizes.

All transfer of parameters is via the stack.

visualSTATE Basic API function	Stack size in bytes	Typical size
SEM_InitSignalQueue	0	0
SEM_InitExternalVariables	0	0
SEM_InitInternalVariables	0	0
SEM_SignalQueuePut	SEM_EVENT_TYPE + 1	2
SEM_SignalQueueGet	2*SEM_EVENT_TYPE	2
SEM_SignalQueueInfo	Pointer	2
SEM_Init	0	0
SEM_InitInstances	1	1
SEM_SetInstance	SEM_INSTANCE_TYPE + 1	2
SEM_Deduct	SEM_EVENT_TYPE + 1. See note ^a	2
SEM_GetOutput	Pointer + 1	3
SEM_GetOutputAll	Pointer + 2* SEM_ACTION_EXPRESSION_TYPE + 2	8

Table 15: Typical stack sizes, Basic API

visualSTATE Basic API function	Stack size in bytes	Typical size
SEM_NextState	2	2
SEM_Inquiry	1	1
SEM_GetInput	2*Pointer + 1	5
SEM_GetInputAll	2*Pointer + 2* SEM_EVENT_TYPE + 2	6
SEM_Name	2*Pointer + SEM_EXPLANATION_TYPE + 3 + 2*short	11
SEM_NameAbs	Pointer + SEM_EXPLANATION_TYPE + 2	4
SEM_Expl	2*Pointer + SEM_EXPLANATION_TYPE + 3 + 2*short	11
SEM_ExplAbs	Pointer + SEM_EXPLANATION_TYPE + 2	4
SEM_State	Pointer + SEM_STATE_MACHINE_TYPE + 1	4
SEM_StateAll	Pointer + 2* SEM_STATE_MACHINE_TYPE + 1	5
SEM_Machine	Pointer + SEM_STATE_TYPE + 1	5
SEM_TableAction	See note ^b	
SEM_ForceState	SEM_STATE_TYPE + 1	3

Table 15: Typical stack sizes, Basic API (Continued)

a. If event parameters are used, SEM_Deduct will use more stack space, depending on the number and types of parameters used.

b. SEM_TableAction in itself does not use any stack. The action expressions called by SEM_TableAction may use stack space, depending on the number and types of parameters used. If for example an action takes two integer parameters, stack usage will be 4 bytes

visualSTATE Expert API function	Stack size in bytes	Typical size
SMP_Expl	3*Pointer + 2*short + 1 long + 1 int + SEM_EXPLANATION_TYPE + 1	17
SMP_ExplAbs	2*Pointer + SEM_EXPLANATION_TYPE + long + 2	10
SMP_ForceState	Pointer + SEM_STATE_TYPE + 1	5
SMP_Inquiry	Pointer + 1	3
SMP_InitInstances	Pointer + long + 1	7
SMP_SetInstance	2*Pointer + SEM_INSTANCE_TYPE + SEM_STATE_MACHINE_TYPE + 1	7
SMP_Machine	2*Pointer + SEM_STATE_TYPE + 1	7

Table 16: Typical stack sizes, Expert API

visualSTATE Expert API function	Stack size in bytes	Typical size
SMP_Connect	6*Pointer + 3	15
SMP_Free	Pointer	2
SMP_Init	Pointer + SEM_STATE_MACHINE_TYPE + 1	5
SMP_InitGuardCallBack	3*Pointer	6
SMP_InitSignalDBCallBack	2*Pointer + SEM_EVENT_TYPE	5
SMP_InitSignalQueue	Pointer	2
SMP_SignalQueuePut	Pointer + SEM_EVENT_TYPE + 1	4
SMP_SignalQueueGet	Pointer + 2*SEM_EVENT_TYPE	4
SMP_Deduct	Pointer + SEM_EVENT_TYPE + 1 See note ^a	4
SMP_GetOutput	2*Pointer + SEM_STATE_TYPE + SEM_RULE_DATA_TYPE + SEM_STATE_MACHINE_TYPE + 5 + [if guard conditions or signals: 2]	13 (15)
SMP_GetOutputAll	2*Pointer + 2*SEM_ACTION_EXPRESSION_TYPE + 2 + SMP_GetOutput	10 + 13 (15)
SMP_NextState	Pointer + SEM_STATE_MACHINE_TYPE + SEM_ACTION_EXPRESSION_TYPE + 2	7
SMP_Name	3*Pointer + SEM_EXPLANATION_TYPE + int + long + short + 3	17
SMP_NameAbs	2*Pointer + SEM_EXPLANATION_TYPE + long + 2	10
SMP_StateAll	2*Pointer + SEM_STATE_MACHINE_TYPE + 1	7
SMP_TableAction	See note ^b	
SMP_State	2*Pointer + SEM_STATE_MACHINE_TYPE + 1	7

Table 16: Typical stack sizes, Expert API (Continued)

a. If event parameters are used, SMP_Deduct will use more stack space, depending on the number and type of parameters.

b. SMP_TableAction in itself does not use any stack. The action expressions called by SMP_TableAction may use stack space, depending on the number and types of parameters used. If for example an action takes two integer parameters, stack usage will be 4 bytes.

RAM/ROM usage

Using the generated code and the visualSTATE OSEK API has little impact on the normal RAM/ROM usage by the visualSTATE API.

For information on RAM/ROM usage by the visualSTATE standard APIs, refer to *visualSTATE API Guide*.

BASIC API

There is no overhead whatsoever when the visualSTATE Basic API is used together with the visualSTATE OSEK API.

EXPERT API

There is a slight increase in ROM consumption when the visualSTATE Expert API is used together with the visualSTATE OSEK API. There is no increase in RAM consumption.

The visualSTATE OSEK API needs information about each visualSTATE System in order to use the visualSTATE standard API. An array with the size of the total number of visualSTATE Systems will be placed in ROM. Each array element will have the following structure:

```
typedef struct VSSystemSetup {
    void* VSData;
    unsigned char (**VSGuard)(SEM_CONTEXT *Context);
    unsigned char (*VSDeduct)(SEM_CONTEXT *Context, SEM_EVENT_TYPE
EventNo);
    void (**VSAction)(SEM_CONTEXT* Context);
    void (*VSInitExtVar)(void);
    void (*VSInitIntVar)(void);
    SEM_EVENT_TYPE VS_RESET;
} VSSystemSetup;
```

The total increase in ROM usage will depend on the target hardware, target compiler and the size of the visualSTATE System.

Example

A void pointer may require 2 bytes of ROM. If there are more than 256 events, the SEM_EVENT_TYPE also requires 2 bytes. This gives a total of 7 * 2 bytes = 14 bytes ROM overhead per visualSTATE System.

A visualSTATE Project with six visualSTATE Systems gives a total overhead of 84 bytes of ROM.

Part II: General reference

This part of the visualSTATE[®] User Guide includes the following chapters:

- Navigator menu commands
- Designer shortcuts
- Designer menu commands
- Validator shortcut keys
- Validator menu commands.
- Verificator command line options
- Coder options
- Documenter options.





Navigator menu commands

This chapter describes the Navigator menu commands and lists their corresponding shortcut keys and toolbars. The Navigator has the following menus:

- File menu
- Edit menu
- View menu
- Project menu
- Tools menu
- Window menu
- Help menu.

For a description of the Navigator toolbars, see *Navigator toolbars*, page 19.

File menu

This menu contains the following commands:

Command	Shortcut key	Toolbar
New... Creates a new workspace file, Project file, System or Statechart file.	Ctrl+N	Standard
Open... Opens an existing file.	Ctrl+O	Standard
Save Saves the file in the currently active window.	Ctrl+S	Standard

Table 17: Navigator File menu commands

Command	Shortcut key	Toolbar
Save As		
Saves a copy of the file in the currently active window under a new name.		
Close		
Open Workspace		
Opens a visualSTATE workspace.		
Save Workspace		
Saves the current workspace and all its files.		
Save Workspace As...		
Saves a copy of the current workspace under a new name (extension must be <code>.vsw</code>).		
Close Workspace		
Closes the workspace.		
Source Control		
Opens a submenu of source control commands.		
Source control>Get Latest Version...		
Retrieves a read-only copy of the selected source control files.		
Source control>Check Out...		
Retrieves the latest version of the selected file from source control. The version is writable.		
Source control>Check In...		
Checks in a checked-out file and thereby updates the source control system with the changes made in the file.		
Source control>Undo Check Out...		
Cancels a Check Out operation, undoing all changes.		
Source control>Add to source control...		
Copies a file, or group of files, into the source control system.		
Source control>Remove from source control...		
Removes the selected files from the source control system.		
Source control>Refresh Status		
Retrieves the source control status of all files in the Project.		
Source control>Launch Source Control System...		
Launches the source control application you have set up as default for your visualSTATE files.		
Exit	Alt+F4	

Table 17: Navigator File menu commands (Continued)

Edit menu

The commands on this menu are used for editing text in the currently active Navigator window, and correspond to standard Windows editing commands.

View menu

This menu contains the following commands:

Command	Shortcut key	Toolbar
Toolbars		
Opens a submenu for showing or hiding toolbars.		
Status bar		
Shows or hides the status bar.		
Workspace	Alt+0	
Shows or sets focus to the workspace browser.		
Output	Alt+2	
Shows or sets focus to the output window.		
Go To		Internet Browser
Opens a submenu of standard Internet browser commands for browsing the web or local files.		
Stop	Esc	Internet Browser
Stops the current browser search.		
Refresh	F5	Internet Browser
Updates the content of the page in the browser.		
Properties	Alt+Enter	Standard
Opens a box showing the properties of the currently active item.		

Table 18: Navigator View menu commands

Project menu

This menu contains the following commands:

Command	Shortcut key	Toolbar
Designer...	F7	
Launches the Designer application.		
Validator...	F8	
Launches the Validator application.		

Table 19: Navigator Project menu commands

Command	Shortcut key	Toolbar
Code-generate Starts code generation for the selected Project.	F9	Standard
Verify All Systems Verifies one or more Systems in the selected Project.	F10	Standard
Verify System Verifies the System selected in the workspace browser.	Ctrl+F10	
Document Creates a documentation report for the Project selected.	F11	Standard
Options Opens a submenu for setting Coder, Verificator and Documenter options.		
Options>Code generation	Alt+F9	Standard
Options>Verification	Alt+F10	Standard
Options>Documentation	Alt+F11	Standard

Table 19: Navigator Project menu commands (Continued)

Tools menu

This menu contains the following commands:

Command	Shortcut key	Toolbar
OSEK Wizard... Launches the visualSTATE OSEK wizard		
Settings... Opens a dialog box for configuring the Navigator.		
Custom Commands... Opens a custom commands editor.		

Table 20: Navigator Tools menu commands

Window menu

This menu offers basic window display commands such as *Tile* and *Cascade* windows.

Help menu

This menu gives you access to the online help, user documentation PDF files, and general information about the visualSTATE Navigator.

Designer shortcuts

This chapter lists the shortcuts available in the Designer.

General

Create new Project	CTRL+N
Open Project	CTRL+O
Save Project	CTRL+S
Print the design	CTRL+P
Set focus to Project browser	ALT+0
Set focus to element browser	ALT+1
Set focus to property window	ALT+2
Set focus to output window	ALT+2
Update/refresh window	F5
Open Designer online help	F1
Close window	ALT+F4

EDITING

Edit name	F2
Undo the latest action	CTRL+Z
Cut selected text or graphics	CTRL+X
Copy text or graphics	CTRL+C
Paste text or graphics	CTRL+V
Open Compose window for selected object	ENTER
Search for an element	CTRL+F

Diagram tools

GENERAL

Activate selection tool	CTRL+0
Deactivate a tool	Right mouse button
Activate note tool	CTRL+9
Activate zoom tool	CTRL+SHIFT+2
Go to parent diagram	BACKSPACE
To delete an item just added	DELETE

TRANSITIONS

Activate normal transition tool	CTRL+3
Activate curved transition tool	CTRL+ALT+3
Activate orthogonal transition tool	CTRL+SHIFT+3
Activate self- transition tool	CTRL+4
Remove last placed route point	Right mouse button (transition tool must be active)
Clone a route point (add a route point to a transition).	Press CTRL and drag route point
Delete dragged route point	Drag route point onto another route point on the same transition.

STATES

Activate simple state tool	CTRL+2
Activate composite state tool	CTRL+SHIFT+2
Define number of regions in composite state	Press and hold CTRL while drawing a composite state.
Swap two regions within the same state	SHIFT + drag a region
Activate initial state tool	CTRL+5
Activate shallow history state tool	CTRL+ALT+5
Activate deep history state tool	CTRL+SHIFT+5
Activate final state tool	CTRL+6
Activate join state tool	CTRL+7
Activate fork state tool	CTRL+ALT+7
Activate junction state tool	CTRL+SHIFT+7
Activate connector state tool	CTRL+8

Project, System and statechart diagram views

Select object(s)	Click and hold left mouse button on the statechart (not on an object). Drag a rectangle. All objects within the rectangle will be selected when you release the left mouse button.
Clone selected object(s)	Press CTRL and drag selection
Delete selected text or object	DELETE
Cancel editing	ESC

NAVIGATION

Go to next object (from selected object)	TAB
Go to previous object (from selected object)	SHIFT+TAB
Scroll up	CTRL+UP ARROW
Scroll down	CTRL+DOWN ARROW
Scroll left	CTRL+LEFT ARROW
Scroll right	CTRL+RIGHT ARROW
Scroll up one page	PAGE UP / CTRL+SHIFT+UP ARROW
Scroll down one page	PAGE DOWN / CTRL+SHIFT+DOWN ARROW
Scroll left one page	CTRL+PAGE UP
Scroll right one page	CTRL+PAGE DOWN
Go to top of view	HOME
Go to bottom of view	CTRL+HOME
Go to the far left of view	END
Go to the far right of view	CTRL+END
To change between windows and views	CTRL+TAB

MOVING OBJECTS

Move selected objects one grid unit	ARROW keys
Move selected objects one pixel	SHIFT+ARROW keys

ZOOMING STATECHART DIAGRAMS

Zoom in	+
Zoom out	-
Zoom all objects	Zoom "+"
Zoom selection	Zoom "-"
Set zoom percentage to 100%	Press CTRL and click right mouse button (zoom tool must be active)

GRID AND SNAP

Show grid	ALT+G
Grid on top	ALT+SHIFT+G
Use snap	ALT+S

Element browser

Create a new element	CTRL+N
Select next element type	CTRL+PAGE DOWN
Select previous element type	CTRL+PAGE UP
Delete selected element	DELETE

Designer menu commands

This chapter describes the Designer menu commands and lists their corresponding shortcut keys and toolbars. The Designer has the following menus:

- File menu
- Edit menu
- View menu
- Insert menu
- Format menu
- Tools menu
- Window menu
- Help menu.

For a description of the Designer toolbars, see *Designer toolbars*, page 53.

File menu

This menu contains the following commands:

Command	Shortcut key	Toolbar
New... Creates a new Project. The new Project and its Systems can later be imported into a Navigator workspace, for example for testing and code generation by means of the other visualSTATE tools.	Ctrl+N	Standard
Open Project... Opens an existing visualSTATE Project file (extension <code>vsp</code>).	Ctrl+O	Standard
Save Project Saves the current Project and all its files.	Ctrl+S	Standard

Table 21: Designer File menu commands

Command	Shortcut key	Toolbar
Close Project Closes the Project file.		Standard
Save As... Saves the selected Statechart file under a new name.		
Import... Imports function declarations and constants contained in a C header file.		Standard
Page Setup... This command is used for defining headers, footers, margins etc. for print-out of statechart diagrams.		Standard
Print Preview... Opens a view showing how the statechart diagrams will look when printed. It is possible to print from the view.		Standard
Print... Prints the current statechart diagrams.	Ctrl+P	Standard
Source Control Opens a submenu of source control commands. The commands correspond to the source control commands on the Navigator File menu (see <i>File menu</i> , page 337).		Source control
Exit Closes the visualSTATE Designer application.		

Table 21: Designer File menu commands (Continued)

Edit menu

Figure 217, page 346 shows the commands available on this menu which are described in Table 22, page 347.

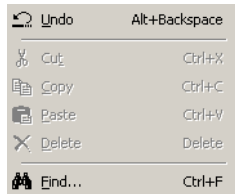


Figure 217: Designer Edit menu

Command	Shortcut key	Toolbar
<p>Undo</p> <p>Undoes the last action performed, such as Move, Delete, Rename etc. The type of Undo is specified in the menu, for example Undo Move. Undo depth is specified by choosing Tools>Settings.</p>	Alt+Backspace	Standard
<p>Cut</p> <p>Removes the selected item (state or text) and places it on the clipboard. The item may then be pasted into another field or diagram. The item will remain on the clipboard until it is replaced by another item. It may be pasted more than once.</p>	Ctrl+X	Standard
<p>Copy</p> <p>Makes a copy of the selected item (state or text) and places it on the clipboard. The item may then be pasted into another field or diagram. The item will remain on the clipboard until it is replaced by another item. It may be pasted more than once.</p>	Ctrl+C	Standard
<p>Paste</p> <p>Inserts a copy of the item from the clipboard.</p> <p>Note: It is not possible to cut/copy/paste between different types of views.</p>	Ctrl+V	Standard
<p>Delete</p> <p>Deletes the selected object(s).</p>	Delete	
<p>Find</p> <p>Opens a window where you can search for element types in Projects, Systems or topstates.</p>	Ctrl+F	

Table 22: Designer Edit menu commands

View menu

This menu allows you to hide or show windows and toolbars by clicking the individual commands on the menu (see *Designer toolbars*, page 53). *Figure 218*, page 348 shows the commands available on this menu which are described in *Table 23*, page 348.

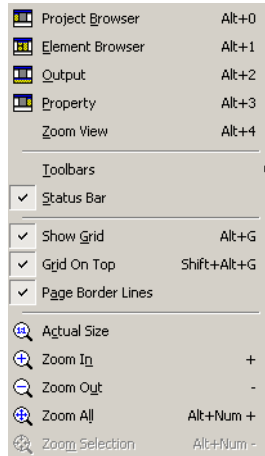


Figure 218: Designer View menu

Command	Shortcut key	Toolbar
Project Browser Shows or hides the Project Browser window.	Alt+0	
Element Browser Shows or hides the Element Browser window.	Alt+1	
Output Shows or hides the output window.	Alt+2	
Property Shows or hides the property window.	Alt+3	
Zoom view Opens the zoom view.	Alt+4	
Toolbars Choosing this command opens a submenu by which you can select display of the Standard, Diagram, Size, Source Control and Zoom toolbars.		
Status bar Shows or hides the status bar. The status bar is placed at the bottom of the screen and displays information about the menu commands, toolbar commands and cursor position in diagram windows.		

Table 23: Designer View menu commands

Command	Shortcut key	Toolbar
Show Grid Shows or hides grid.	Alt+G	
Grid On Top Choose this command to have a grid drawn over all other elements in the Diagram view.	Shift+Alt+G	
Page Border Lines Shows or hides page border lines. The lines define the editable area of the diagram page.		
Actual Size Resets size (zoom percentage is reset to 100).		Zoom
Zoom In Enlarges the size of the items in the current statechart diagram window.	+	Zoom
Zoom Out Reduces the size of the items in the current statechart diagram window.	-	Zoom
Zoom All Zooms so that all objects in the statechart diagram become visible.	Alt + Num +	Zoom
Zoom Selection Zooms so that the selected objects in the statechart diagram become visible.	Alt + Num -	Zoom

Table 23: Designer View menu commands (Continued)

Insert menu

On this menu you can select the drawing tools that are also available on the Diagram toolbar. *Figure 219*, page 350 shows the commands available on this menu.

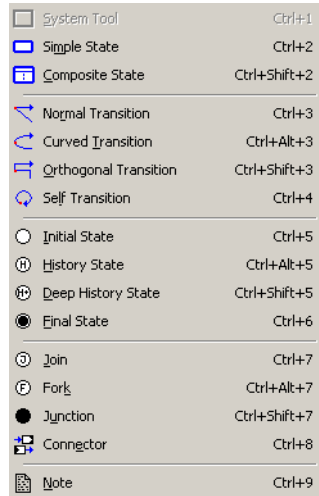


Figure 219: Designer Insert menu

Note: The System Tool is only available when the Project Browser window is active.

Format menu

Figure 220, page 350 shows the commands available on this menu which are described in Table 24, page 351.

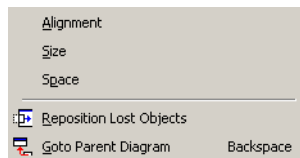


Figure 220: Designer Format menu

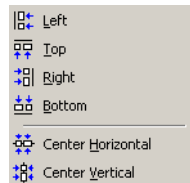


Figure 221: Alignment menu commands, Designer Format menu

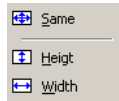


Figure 222: Size menu commands, Designer Format menu

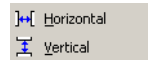


Figure 223: Space menu commands, Designer Format menu

Command	Shortcut key	Toolbar
Alignment		
Opens a submenu of object alignment commands (also found on the Size toolbar). Objects must be selected for the commands to be available (click on the states while at the same time pressing the SHIFT key down). Objects will be aligned according to the state last selected.		
Size		
Opens a submenu of object size commands (also found on the Size toolbar). Objects must be selected for the commands to be available. Sizes of objects will be changed according to the object last selected.		
Space		
Opens a submenu of commands for distributing space evenly between objects (also found on the Size toolbar). At least three objects must be selected for the commands to be available. Objects are distributed according to the space between the three objects last selected.		
Reposition Lost Objects		
Repositions objects located outside the diagram onto the diagram.		
Go to Parent Diagram		
Moves your diagram one level up in the hierarchy, for example from statechart diagram view to System view.		

Table 24: Designer Format menu commands

Tools menu

On this menu you specify various settings for your diagrams and their elements, for example grid size and zoom percentage. *Figure 224*, page 352 shows the commands available on this menu which are described in *Table 25*, page 352.

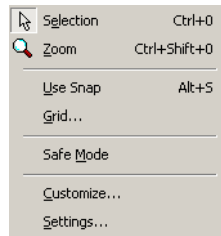


Figure 224: Designer Tools menu

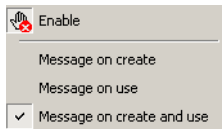


Figure 225: Safe Mode menu commands, Designer Tools menu

Command	Shortcut key	Toolbar
Selection Activates/deactivates selection tool. Choosing the command when using a Diagram tool, for example Insert State, deactivates the Diagram tool.	Ctrl+O	Diagram
Zoom Activates/deactivates the zoom tool.	Ctrl+Shift+O	Zoom
Use Snap Activates/deactivates snap. When objects are moved, snap will move the top-left corners of selected objects to the corners of the grid.	Alt+S	
Grid... Opens a grid setup dialog box.		

Table 25: Designer Tools menu commands

Command	Shortcut key	Toolbar
Safe Mode		Standard
Opens a submenu of safe mode commands. By enabling Safe Mode you will receive a warning when you create or use a non-verifiable element, according to selection on the submenu.		
Customize...		
Opens a dialog box for specifying the default look of the objects drawn in statechart diagrams, for example default state color.		
Settings...		
Opens a Designer configuration dialog box for specifying backup option, undo depth, display of messages, etc.		

Table 25: Designer Tools menu commands (Continued)

Window menu

This menu offers basic window display commands such as *Tile* and *Cascade* windows. See *Figure 226*, page 353.

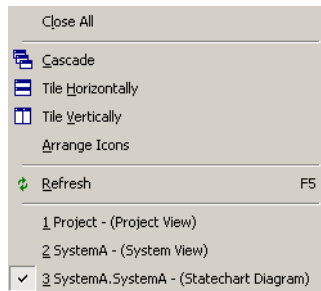


Figure 226: Designer Window menu

Help menu

This menu gives you access to the online help (shortcut key: F1) and general information about the visualSTATE Designer. See *Figure 227*, page 353.

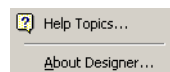


Figure 227: Designer Help menu

Validator shortcut keys

This chapter lists the shortcut keys available in the Validator.

General

Create a new workspace	CTRL+N
Open a workspace	CTRL+O
Save an open file (if the active window is a Test Sequence File window).	CTRL+S
Exit the Validator	ALT+F4
Open the Validator online help	F1
Stop timer (focus must be set to Timers window)	DELETE

Windows

Open a new System window	CTRL+1
Open a new Event window	CTRL+2
Open a new Action window	CTRL+3
Open a new Variable window	CTRL+4
Open a new Guard Expression window	CTRL+5
Open a new Signal Queue window	CTRL+6
Show runtime model (only when in target mode)	ALT+F8
Show and/or set focus to the Field Chooser window	ALT+0
Show and/or set focus to the System Setup window	ALT+1
Show and/or set focus to the output window	ALT+2
Show and/or set focus to the Watch window	ALT+3
Show and/or set focus to the Timer window	ALT+4
Show and/or set focus to the Breakpoints window	ALT+9

Editing

Undo the latest action	CTRL+Z
Open the Edit Breakpoints dialog box	ALT+F9

Debugging

Initialize System(s)	ALT+I
Start playing a recorded test sequence	F9
Step recorded test sequence one step forward	F10
Start execution of a recorded test sequence and stop it again at the selected step (Run to cursor)	CTRL+F10
Set the selected step as the next step in the recorded test sequence	ALT+F10
Stop execution of recorded test sequence and return the cursor to the first step	SHIFT+F5
Pause execution of recorded test sequence	CTRL+F5
Mark the selected step of a test sequence file as a stop point	CTRL+F9
Start recording to a test sequence file	ALT+R
Send the first signal in the queue	F11
Switch between automatic and manual emptying of signal queue	SHIFT+F11
Empty all signal queues by sending the signals	CTRL+F11
Start static or dynamic analysis (depending on the active window)	CTRL+F8
Add element to Watch window	SHIFT+F9

Navigation in test sequence files

Go to the next sequence	CTRL+ DOWN ARROW
Go to the previous sequence	CTRL+UP ARROW

Validator menu commands

This chapter describes the Validator menu commands and lists their corresponding shortcut keys and toolbars. The Validator has the following menus:

- File menu
- Edit menu
- View menu
- Debug menu
- RealLink menu
- Altia menu
- Window menu
- Help menu.

For a description of the Validator toolbars, see *Validator toolbars*, page 158.

File menu

Figure 228, page 358 shows the commands available on this menu which are described in Table 26, page 358.

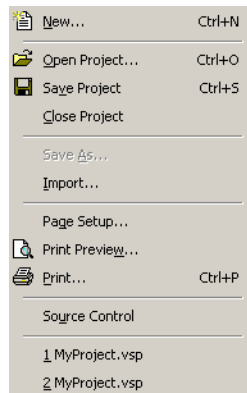


Figure 228: Validator File menu

Command	Shortcut key	Toolbar
New Workspace Creates a new Validator workspace. You will be prompted to specify whether or not to load a visualSTATE Project into the new workspace. If you load a Project into the workspace, the Validator will automatically open and arrange the windows according to Classic Simulation style, giving a good starting point for simulation.	Ctrl+N	Standard
Open Workspace... Opens an existing Validator workspace.	Ctrl+O	Standard
Close Workspace Closes the open workspace. If this is a new workspace not yet assigned to a file, you will be prompted to specify whether or not to save the workspace. If the workspace has been assigned to a file, and anything regarding the functionality of the workspace has changed since the last save, you will also be prompted to specify whether or not to save the workspace. If the workspace has been assigned to a file, window setup etc. will always be saved.		

Table 26: Validator File menu commands

Command	Shortcut key	Toolbar
Save Workspace		Standard
Saves the current workspace. Functionality setup and window setup of the workspace will be saved.		
Save Workspace As...		Standard
Opens a Save As dialog box by which the workspace can be saved under a new name.		
Load Project...		
Loads a visualSTATE Project into the Validator workspace. Before interactive simulation, automatic simulation and analysis can be performed, a visualSTATE Project must be loaded into the workspace. The Validator workspace only handles one visualSTATE Project at a time.		
Close Project		
Closes the visualSTATE Project loaded. Closing a Project will remove all Project-related information from the Validator workspace, including breakpoints, System setup, and Windows directly related to the specific Project.		
During the closing of a Validator workspace, the Project is automatically closed, so unless another Project is to be loaded into the current workspace, there is no need to close the Project.		
Test Sequence File		
Opens a submenu of commands for the handling of test sequence files.		
Test Sequence File>New		
Opens a new test sequence file. No file is created before save.		
Test Sequence File>Open		
Opens an existing test sequence file thereby making it possible to append commands to a sequence, create a new sequence etc.		
Test Sequence File>Close		
Closes an open test sequence file.		
Test Sequence File>Save		
Saves an open test sequence file.		
Test Sequence File>Save As...		
Saves an open test sequence file under a new name.		
Analysis		
Opens a submenu of commands for the handling of static and dynamic analysis files.		

Table 26: Validator File menu commands (Continued)

Command	Shortcut key	Toolbar
Analysis>New Dynamic Creates a new dynamic analysis file and opens it in a new window.		
Analysis>New Static Creates a new static analysis file and opens it in a new window.		
Analysis>Open... Opens either an existing static analysis file or an existing dynamic analysis file. The Validator will by itself resolve file type on the basis of the file extension.		
Analysis>Close Closes an open analysis file.		
Analysis>Save Saves an open analysis file.		
Analysis>Save As... Saves an open analysis file under a new name.		
Print Prints an open Validator file. Files that can be printed are test sequence files, and static and dynamic analysis files.		Standard
Most Recently Used Files list Contains the four most recently used Validator workspaces. The first time the Validator is started, the list is empty.		
Exit Closes the Validator application. It is only possible to close the Validator application if it is not performing any form of analysis or executing recorded test sequences.	Alt+F4	

Table 26: Validator File menu commands (Continued)

Edit menu

Figure 229, page 360 shows the commands available on this menu which are described in Table 27, page 361.

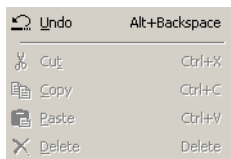


Figure 229: Validator Edit menu

Command	Shortcut key	Toolbar
<p>Undo</p> <p>All operations on the Systems in the loaded visualSTATE Project can be undone. The commands correspond to the commands that can be recorded to a file and played from a test sequence file.</p> <p>Note: The Undo command only applies to the Validator model, not the runtime model (RealLink).</p> <p>The following commands can be undone:</p> <ul style="list-style-type: none"> Initializing a System. Sending an event. Sending a signal. Setting the value for an internal variable. Setting the value for an external variable. Setting the return value for an action. Forcing a System into a specific state. <p>Although the manual sending of an event will cause the event to be sent to all enabled Systems, undoing the Send Event command will only undo the event last sent. To undo all Send Event commands, activate the Undo command the same number of times as the number of enabled Systems in the Project.</p> <p>Global events can be sent to multiple Systems. To undo the sending of a global event, you must apply Undo as many times as there are Systems to which the event was sent.</p> <p>If the Validator is recording to a test sequence file, the Undo command will cause the last recorded command to be removed from the test sequence file.</p> <p>If the command that is being undone is performed from a test sequence file, the Undo command will cause the recorded test sequence to be reversed to its original position.</p> <p>Designer Path</p> <p>This command specifies where the Validator is to locate visualSTATE Designer. This information must be specified in the Validator if Graphical Animation is to be performed.</p> <p>The command launches a Windows Open File dialog box to enable navigation through the available drives and directories of the System. If the visualSTATE software is installed in its default directories, you do not have to specify where to locate the visualSTATE Designer.</p>	Ctrl+Z	Debug

Table 27: Validator Edit menu commands

Command	Shortcut key	Toolbar
Speed		
Opens a submenu of commands related to execution speed for test sequence files.		
Timer Tick Length		
Opens a submenu of commands by which it is possible to set the tick length of timer ticks used in the Validator. You can set the tick length to one of four predefined values, or set up a custom tick length.		
Breakpoints...	Alt+F9	
Opens a dialog box for defining breakpoints.		

Table 27: Validator Edit menu commands (Continued)

View menu

This menu allows you to hide or show windows and toolbars. *Figure 230*, page 362 shows the commands available on this menu which are described in *Table 28*, page 362.

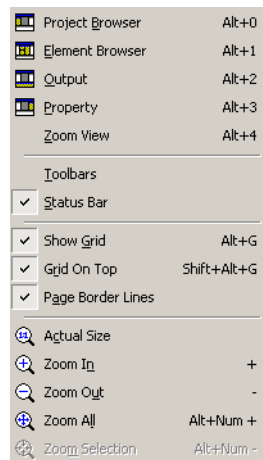


Figure 230: Validator View menu

Command	Shortcut key	Toolbar
Field Chooser	Alt+0	
Shows and/or sets focus to the Field Chooser window.		

Table 28: Validator View menu commands

Command	Shortcut key	Toolbar
System Setup Shows and/or sets focus to the System Setup window. System setup is only relevant if the visualSTATE Project contains more than one System and/or the System contains multiple instances.	Alt+1	
Output Shows and/or sets focus to the output window.	Alt+2	
Watch Shows and/or sets focus to the Watch window to which elements from other windows can be added.	Alt+3	
Timers Shows and/or sets focus to the Timer window. The Timer window shows all running timers.	Alt+4	
Breakpoints Shows and/or sets focus to the Breakpoints window.	Alt+9	
Standard Shows or hides the Standard toolbar.		
Debug Shows or hides the Debug toolbar.		
RealLink Shows or hides the RealLink toolbar.		
Analyze Shows or hides the Analysis toolbar.		
Status Bar Shows or hides the status bar.		

Table 28: Validator View menu commands (Continued)

Debug menu

Figure 231, page 364 shows the commands available on this menu which are described in Table 29, page 364.

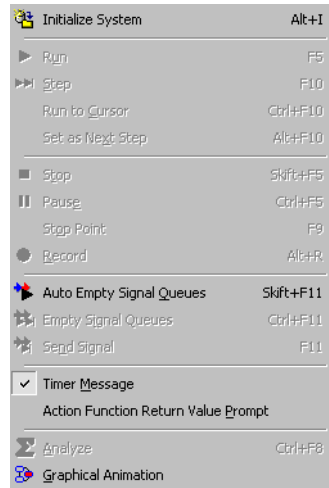


Figure 231: Validator Debug menu

Command	Shortcut key	Toolbar
Initialize System / Initialize System(s) Initializes the System (or Systems, if the Project contains multiple Systems) to its startup state. The command does the following: <ul style="list-style-type: none"> ● Initializes the state configuration to State-Undefined. ● Initializes all internal and external variables to their initial values. ● Resets the signal queue. 	Alt+I	Debug
Run Executes a recorded test sequence.	F9	Debug
Step Steps a test sequence one step forward.	F10	Debug
Run to Cursor This command sets a step in a recorded test sequence as a temporary stop point.	Ctrl+F10	
Set as Next Step Allows jumping back and forward in a recorded test sequence.	Alt+F10	
Stop Stops the playing of a recorded test sequence and resets the cursor to the start of the file.	Shift+F5	Debug

Table 29: Validator Debug menu commands

Command	Shortcut key	Toolbar
<p>Pause</p> <p>Stops the playing of a test sequence file and leaves the cursor at the current step.</p>	Ctrl+F5	Debug
<p>Stop Point</p> <p>Sets a breakpoint in a recorded test sequence. The command is used to stop execution at a critical point.</p>	Ctrl+F9	
<p>Record</p> <p>Starts or stops recording to a test sequence file.</p>	Alt+R	Debug
<p>Auto Empty Signal Queues</p> <p>Turns the Auto Signal Queue mode of the Validator workspace on/off. When Auto Signal Queue mode is applied, the signal queue is automatically emptied when an event is sent manually to a System. During execution, the signal queue is not emptied automatically.</p>	Shift+F11	Debug
<p>Empty Signal Queues</p> <p>Sends the first signal in the first queue. If the queue still contains signals, the next signal is sent. The command continues to send signals until the queue is empty. It then moves on to the next System having a signal queue. The command will continue processing until all signal queues in the Systems are empty.</p>	Ctrl+F11	Debug
<p>Send Signal</p> <p>Sends the top signal in the first queue not empty. The order in which the queues are emptied is determined by the System setup.</p>	F11	Debug
<p>Timer Message</p> <p>This command specifies whether or not a warning message should be given whenever an event from a timer is about to be sent.</p>		
<p>Action Function Return Value Prompt</p> <p>Choose this command if you want to be prompted for action function return values.</p>		
<p>Analyze</p> <p>This command starts a static or dynamic analysis whichever is the active window.</p>	Ctrl+F8	Analysis
<p>Graphical Animation</p> <p>This command launches the visualSTATE Designer and establishes a link between the Designer and Validator applications. This gives a graphical view of the simulation of the System.</p>		Debug

Table 29: Validator Debug menu commands (Continued)

RealLink menu

Figure 232, page 366 shows the commands available on this menu which are described in Table 30, page 366.



Figure 232: Validator RealLink menu

Command	Shortcut key	Toolbar
Connect/ Disconnect Connect establishes connection between the Validator and target. Disconnect will set the target into Run mode and close the RealLink connection.	F6	RealLink
Reset Communication Resets the communication to its initial state. This button is only active when the Validator is communicating with the target.		RealLink
Run This command will cause the target to run in real-time mode. This means that the Validator does not update its windows to reflect the status of the target. To stop Run mode, click the Break button.	F8	RealLink
Autostep First performs a macrostep, then retrieves data from the target. This sequence of actions is repeated.	Shift+F8	RealLink
Macrostep Performs a macrostep.	F7	RealLink
Microstep Performs a microstep.	Shift+F7	RealLink
Break Stops the target.	Shift+F6	RealLink

Table 30: Validator RealLink menu commands

Command	Shortcut key	Toolbar
Properties		
Used for selection and configuration of a communication module for RealLink.		

Table 30: Validator RealLink menu commands (Continued)

Altia menu

Figure 233, page 367 shows the commands available on this menu which are described in Table 31, page 367. A detailed description of the use of the Validator Altia facility is given in *Prototyping with Altia*, page 279.

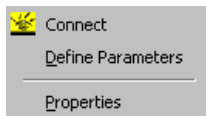


Figure 233: Validator Altia menu

Command	Shortcut key	Toolbar
Connect		Debug
Connects the Validator to an Altia design.		
Define Parameters		
Opens a dialog box for defining Altia parameter values for visualSTATE events and action functions.		
Properties		
Opens a dialog box where you can configure the Altia connection.		

Table 31: Validator Altia menu commands

Window menu

Figure 234, page 368 shows the commands available on this menu which are described in Table 32, page 368.

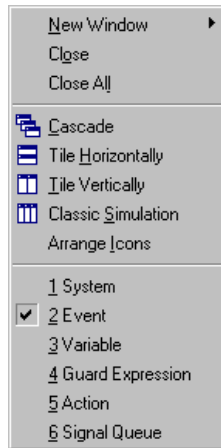


Figure 234: Validator Window menu

Command	Shortcut key	Toolbar
New Window		
Opens a submenu of commands by which a number of windows can be opened.		
New Window>System	Ctrl+1	
Opens a new System window.		
New Window>Event	Ctrl+2	
Opens a new Event window.		
New Window>Action	Ctrl+3	
Opens a new Action window.		
New Window>Variable	Ctrl+4	
Opens a new Variable window.		
New Window>Guard Expression	Ctrl+5	
Opens a new Guard Expression window.		
New Window>Signal Queue	Ctrl+6	
Opens a new Signal Queue window.		

Table 32: Validator Window menu commands

Command	Shortcut key	Toolbar
Close Closes the currently active window.		
Close All Closes all open windows.		
Cascade Cascades all open windows.		Standard
Tile Horizontally/ Vertically These commands tile the open windows horizontally and vertically.		Standard
Classic Simulation Activation of this command arranges the windows in Classic Simulation style, with an Event window, a System window and an Action window tiled vertically. All other opened windows are minimized.		Standard
Arrange Icons This command arranges all minimized windows at the bottom of the Validator window.		

Table 32: Validator Window menu commands (Continued)

Help menu

This menu gives you access to the online help (shortcut key: F1) and general information about the visualSTATE Validator. See *Figure 235*, page 369. The commands on this menu are also available on the Standard toolbar.

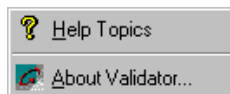


Figure 235: Validator Help menu

Verificator command line options

This chapter describes the options available in the command line version of the Verificator.

General

You can set the Verificator options in the Navigator or via the command line. There is a command line equivalent for all the Verificator options that can be set in the Navigator.

A detailed description of the implications of selecting the various Verificator options is found in *Checks performed by visualSTATE Verificator*, page 123.

Command line syntax

The command line syntax is as follows:

```
Verificator <VS Projectfile> <VS Systemname> [option]...
```

Example 1

```
verificator.exe Example.vsp VS_System -v
```

Explanation: Verify the System `VS_System` in the visualSTATE Project file `Example.vsp` and write the result to the screen.

Example 2

```
verificator.exe Example.vsp VS_System -x local_dead_ends -v  
report.txt -c -s 4
```

Explanation: Verify the System `VS_System` in the Project file `Example.vsp` in compositional mode using a signal queue of length 4. Exclude check for local dead ends. Write the result to the file `report.txt`.

Example 3

```
verificator.exe Example.vsp System -tOut.vlg -dsTopstate.StateA
```

Explanation: Perform a trace for the state `Topstate.StateA`. The Verificator will find a trace to that state if possible and save the resulting trace in the file `Out.vlg`.

List of Verificator command line options

Option	Explanation
-a	Verify in full mode. When verifying in this mode, guard expressions and assignments are included in the verification.
-B<n>	Treat all variables as signed integers encoded in <n> bits.
-c	Perform verification in compositional mode.
-ds<state name>	<state name> is the destination state in trace. This option may be repeated to add more states.
-f	Do <i>not</i> exclude regions and states marked for exclusion from verification. (By default, items marked for exclusion are excluded.)
-g	Verify in guard mode. When verifying in this mode, guard expressions are included in the verification.
-Large -large	Minimize the memory consumption at the expense of a larger time usage. This setting is suitable for large systems.
-p	Use the Verificator options specified in the Navigator.
-s<n>	Verify using a signal queue. This option is followed by a parameter <n>. n is a number, use a signal queue of size n. If the s option is not specified, use a signal queue with the length specified in the visualSTATE Project file.
-S<n>	Specify the initial size of node space. Larger node space usually yields quicker verification. One node uses 20-40 bytes of memory.
-Small -small	Speed up verification at the expense of a larger memory consumption. This setting is suitable for small systems.
-t<trace output>	<trace output> specifies the file the trace should be saved in. Normally with a .vlg extension.
-u	Control variable ranges in assignments. If a range error is detected in an assignment, a fixed constant value is assigned to the variable on the left-hand side. This value does not have to be within the variable's range. The benefit is that constant values can be represented very compactly by the Verificator, and speed up the remaining verification process.
-v[<outfile>]	Writes a text report to the file <outfile>, if specified. If no output file is specified, the text report is written to the screen.
-w	Specify that VS_INT and VS_UINT variables are 16 bits.

Table 33: Verificator command line options

Option	Explanation
-x<check>	Exclude <check>. The <checks> that can be excluded are the following: use activation state_dead_ends local_dead_ends system_dead_ends conflicts

Table 33: Verificator command line options (Continued)

Coder options

This chapter lists the Coder options available and how to set them via the command line. You can also set Coder options via the Navigator as described in *Setting Verificator, Coder and Documenter options*, page 29.

Command line syntax

A Coder option is either a Project option or a System option.

In general, Project options will affect the Project and all Systems contained in it. System options only affect the System(s) for which they are specified.

The command line syntax for Coder options is:

```
<vsp_filename> [--l] [--@<filename>] <--<option>[argument]>*
```

At any point on the command line, the contents of an option file can be inserted. The syntax for specifying an option file is:

```
--@<option-file>
```

The option file must contain options separated by line breaks; thus there is one option on each line. A line is treated as a comment if the line starts with the character sequence `//`.

Specifying `--l` will load options from the specified `vsp` file.

Both Project options and System options can be specified anywhere on the command line. System options that are specified before any System has been specified (option `-v`) apply to all Systems.

If no options and no `vsp` file are specified on the command line, a list of the options will be displayed.

The command line is case-sensitive.

COMMAND LINE EXAMPLES

Example 1

```
Coder Mobile.vsp
```

This command will generate a VS Project located in the file `Mobile.vsp`.

Example 2

```
Coder Mobile.vsp -api_type1 -api_version4 -Vmobile1 -txte3 -txts3
-txta3 -Vmobile2
```

This command will generate a VS Project containing VS Systems named `Mobile1` and `Mobile2`. Code generation will be for Expert API version 4.

In addition, the VS System `Mobile1` will be generated with names and explanation texts for events, states, and action functions.

Example 3

```
Coder Mobile.vsp --@MobileSetup.txt
```

Contents of the `MobileSetup.txt` option file:

```
-Vmobile
-txte3
-txts3
-txta3
```

This command will generate a VS Project containing a VS System named `Mobile`. Code generation will be for Basic API version 4.

In addition, the VS System `Mobile` will be generated with names and explanation texts for events, states, and action functions.

Lists of Coder options

The individual Coder options are listed in *Project options*, page 377 and *System options*, page 386. The contents of the lists correspond to the online help of the Coder Options dialog box (see *Online help*, page 31).

CODER OPTION TYPES

Enumerated options	[E]
Integral options	[I]
Text options	[T]
Boolean options	[B]

Project options

<i>Project option, configuration</i>	<i>Explanation / [option type]</i>
-api_type<argument>	API type. [E] Default argument: Basic Specifies the runtime API to use for code generation. 0 (Basic): The Basic API is the most efficient API in all respects, but it can only handle a VS Project with a single VS System and it cannot load VS Projects from disk. 1 (Expert): The Expert API should primarily be used for VS Projects containing multiple VS Systems, but it can also be used for VS Projects containing a single VS System. The Expert API must also be used if a VS Project is to be loaded from disk, even if the VS Project only contains a single VS System.
-api_version<argument>	API version. [E] Default argument: Version 4/5 Specifies the version of the runtime API to generate code for. 4 (Version 4/5): Forces generation of code compatible with versions 4 and 5 of visualSTATE.
-reallinkmode<argument>	RealLink mode. [E] Default argument: None Specifies the RealLink mode to generate code for. 0 (None): Disables all RealLink related code generation. 1 (Table-based): Generates RealLink related tables.
-expertDLL[{{0 1}}	Generate for Expert DLL. [B] Default argument: 0 Specifies whether to generate code for the Expert DLL.
-cpp[{{0 1}}	C++ code generation. [B] Default argument: 0 Specifies whether to generate C++ code.
-warnings_are_errors[{{0 1}}	Treat warnings as errors. [B] Default argument: 0 Specifies whether to treat warnings as errors. If set, warnings will be reported as errors.
-warnings_affect_exit_code[{{0 1}}	Warnings affect exit code. [B] Default argument: 0 Specifies whether warnings affect the exit code. If set, warnings will result in an exit code different from 0. This option is primarily intended for command line usage.
-no_warnings[{{0 1}}	Ignore warnings. [B] Default argument: 0 Specifies whether to ignore warnings. If set, warnings will not be reported and cannot affect the exit code.

Table 34: Configuration project options

Project option, configuration (Continued)	Explanation / [option type]
<code>-include_excluded[{{0 1}}</code>	<p>Include excluded items. [B]</p> <p>Default argument: 0</p> <p>Specifies whether to ignore exclusion marks in the design and include all states and regions when generating code.</p>

Table 34: Configuration project options (Continued)

Project option, file output	Explanation / [option type]
<code>-path[driveDir]</code>	<p>Output path. [T]</p> <p>Default argument: coder\</p> <p>Specifies the output path for all generated Project files. If the path does not exist, it is created automatically. The path may be a relative path.</p>
<code>-R[pathname]</code>	<p>Report file. [T]</p> <p>Default argument: vsocoder.cre</p> <p>Specifies the name of the report file. The file contains a header identifying the VS Project, applied options, model characteristics and statistics as well as a summary of the overall code generation. If this option is specified without an argument (file name), the file is not generated.</p>
<code>-geventh[pathname]</code>	<p>Event header file. [T]</p> <p>Default argument: \$PRJ\$_PEvent.h</p> <p>Specifies the name of the file containing VS Project level event definitions. This option is only available if the expert API has been selected and also requires that the option to print event names is set. If an empty argument is specified for this option, the default name is used.</p>
<code>-gextvarh[pathname]</code>	<p>External variable header file. [T]</p> <p>Default argument: \$PRJ\$_PExtVar.h</p> <p>Specifies the name of the file containing VS Project level external variable declarations.</p>
<code>-gextvarc[pathname]</code>	<p>External variable source file. [T]</p> <p>Default argument: \$PRJ\$_PExtVar.c</p> <p>Specifies the name of the file containing VS Project level external variable definitions.</p>
<code>-G[pathname]</code>	<p>Constant header file. [T]</p> <p>Default argument: \$PRJ\$_PCConstant.h</p> <p>Specifies the name of the file containing VS Project level constant definitions.</p>

Table 35: File output project options

Project option, code	Explanation / [option type]
-rdfm<argument>	<p>Rule data format. [E] Default argument: Optimized Specifies the rule data format to use.</p> <p>0 (Optimized): Uses the most optimal rule data format. The Coder determines the optimal rule data format with regard to minimal usage of constant data (size optimization).</p> <p>0 (Format 0): Uses rule data format 0. This format uses 8-bit access to rule data. The format supports rules with a maximum of 15 8-bit elements of each type, but does not support guard expressions and signals.</p> <p>1 (Format 1): Uses rule data format 1. This format uses 8-bit access to rule data. The format supports rules with a maximum of 15 8-bit elements of each type.</p> <p>2 (Format 2): Uses rule data format 2. This format uses 8-bit access to rule data. The format supports rules with a maximum of 255 8-bit elements of each type, but does not support guard expressions and signals.</p> <p>3 (Format 3): Uses rule data format 3. This format uses 8-bit access to rule data. The format supports rules with a maximum of 255 8-bit elements of each type.</p> <p>4 (Format 4): Uses rule data format 4. This format uses 16-bit access to rule data. The format supports rules with a maximum of 15 16-bit elements of each type, but does not support guard expressions and signals.</p> <p>5 (Format 5): Uses rule data format 5. This format uses 16-bit access to rule data. The format supports rules with a maximum of 15 16-bit elements of each type.</p> <p>6 (Format 6): Uses rule data format 6. This format uses 16-bit access to rule data. The format supports rules with a maximum of 255 16-bit elements of each type, but does not support guard expressions and signals.</p> <p>7 (Format 7): Uses rule data format 7. This format uses 16-bit access to rule data. The format supports rules with a maximum of 255 16-bit elements of each type.</p> <p>8 (Format 8): Uses rule data format 8. This format uses 32-bit access to rule data. The format supports rules with a maximum of 255 32-bit elements of each type, but does not support guard expressions and signals.</p> <p>9 (Format 9): Uses rule data format 9. This format uses 32-bit access to rule data. The format supports rules with a maximum of 255 32-bit elements of each type.</p>

Table 36: Code project options

Project option, code (Continued)	Explanation / [option type]
<code>-D<argument></code>	<p>Data width. [E] Default argument: Optimized Specifies the data width for SEM variable types. 0 (Optimized): Optimizes the data widths for SEM type definitions. Selecting this value sets the width for all SEM types to the smallest possible size in order to reduce usage of variable and constant data. 0 (8-bit): Sets the data width of all SEM types to 8-bit. If the intended target handles 8-bit access well, speed will probably be increased. 1 (16-bit): Sets the data width of all SEM types to 16-bit. If the intended target handles 16-bit access well, speed will probably be increased. 2 (32-bit): Sets the data width of all SEM types to 32-bit. If the intended target handles 32-bit access well, speed will probably be increased.</p>
<code>-iev<argument></code>	<p>External variable initialization. [E] Default argument: By definition Specifies the method(s) for external variable initialization. 0 (By function): Initializes variables in a function. If variables need to be reinitialized at some point during execution of the VS model, select this value for the option. For example, this could be the case if VS Systems are reinitialized. 1 (By definition): Initializes variables along with their definition. If variables only need to be initialized once, select this value for the option. 2 (Both): Initializes variables in a function and by definition. This value for the option should only be selected for debug purposes, since one of the methods should suffice.</p>
<code>-iiv<argument></code>	<p>Internal variable initialization. [E] Default argument: By definition Specifies the method(s) for internal variable initialization. 0 (By function): Initializes variables in a function. If variables need to be reinitialized at some point during execution of the VS model, select this value for the option. For example, this could be the case if VS Systems are reinitialized. 1 (By definition): Initializes variables along with their definition. If variables only need to be initialized once, select this value for the option. 2 (Both): Initializes variables in a function and by definition. This value for the option should only be selected for debug purposes, since one of the methods should suffice.</p>
<code>-iss{{0 1}}</code>	<p>Explicitly initialize static storage with zero values. [B] Default argument: 0 Specifies whether explicitly to initialize static storage with zero values (external and internal variables). If the initial value of an external or internal variable is zero, there is no need for an explicit initializer, since the target compiler does the initialization anyway. The option should only be applied for compilers that do not adhere to the standard, i.e. compilers that do not do this initialization as required.</p>

Table 36: Code project options (Continued)

Project option, code (Continued)	Explanation / [option type]
<code>-funcexph<argument></code>	<p>Functional expression handling. [E]</p> <p>Default argument: Function pointer tables</p> <p>Specifies how functional expressions (guard expressions and action expressions) are handled.</p> <p>0 (Function pointer tables): Uses a function pointer table for functional expressions. The table ensures constant time access to functional expressions by defining separate functions for functional expressions and including pointers to those functions in an array.</p> <p>1 (Binary if-else construct): Uses a binary if-else construct for functional expressions. A single function is generated with a binary if-else construct to determine the functional expression to execute. This method should only be used if the compiler does not handle the alternatives efficiently.</p> <p>2 (Switch-case construct): Uses a switch-case construct for functional expressions. A single function is generated with a switch-case construct to determine the functional expression to execute. If the compiler is able to recognize the switch-case construct and convert it into a jump table, this may be the most efficient way to handle functional expressions.</p>
<code>-osm[{{0 1}}]</code>	<p>Optimizes states and state machines. [B]</p> <p>Default argument: 1</p> <p>Specifies whether to optimize states and state machines. If the option is set, any state machine with a single state is optimized away, thus reducing the number of states, the number of state machines and the size of the core model logic.</p>
<code>-useegti[{{0 1}}]</code>	<p>Forces the use of egti tables. [B]</p> <p>Default argument: 0</p> <p>Specifies whether to force the use of event group table indexing. This option is primarily provided for internal use.</p>
<code>-gdef[{{0 1}}]</code>	<p>Generate global definitions. [B]</p> <p>Default argument: 0</p> <p>Specifies whether to generate global definitions, which are used internally by some visualSTATE applications.</p>
<code>-gds[{{0 1}}]</code>	<p>Generate digital signature. [B]</p> <p>Default argument: 0</p> <p>Specifies whether to include a digital signature in the generated code.</p>
<code>-useheap[{{0 1}}]</code>	<p>Use heap memory. [B]</p> <p>Default argument: 1</p> <p>Specifies whether to use heap memory. If heap memory is not used, all variable data except for stack data are allocated statically, and the standard functions malloc and free are not used.</p>

Table 36: Code project options (Continued)

Project option, style	Explanation / [option type]
-style<argument>	Naming style. [E] Default argument: New Specifies which naming style to use. 0 (New): Disables version 3 compatible names for API functions. (All other argument to -style are now disabled.)
-tsem<argument>	SEM type definitions. [E] Default argument: As macros Specifies how to define SEM type definitions. 0 (As typedefs): Uses the 'typedef' keyword for type definitions. Select this value whenever possible, since it helps the compiler to do type checking. 1 (As macros): Uses the '#define' keyword for type definitions. The value is needed for compilers that cannot determine that two type definitions actually are the same.
-tvsvt<argument>	VS type definitions. [E] Default argument: As macros Specifies how to define VS type definitions. 0 (As typedefs): Uses the 'typedef' keyword for type definitions. Select this value whenever possible, since it helps the compiler to do type checking. 1 (As macros): Uses the '#define' keyword for type definitions. The value is needed for compilers that cannot determine that two type definitions actually are the same.

Table 37: Style project options

Project option, extended keywords	Explanation / [option type]
-c51vs_prj<argument>	C51 variable segment. [E] Default argument: None Specifies where to place C51 variable data. If an extended keyword is specified for a specific type of variable data, the C51 segment will be ignored. 0 (None): No C51 keywords are used. Select this value if the intended target does not support C51 keywords. 1 (DATA segment): Coder-generated variable data are stored in DATA. 2 (IDATA segment): Coder-generated variable data are stored in IDATA. 3 (PDATA segment): Coder-generated variable data are stored in PDATA. 4 (XDATA segment): Coder-generated variable data are stored in XDATA.
-kw_context[string]	Ext. keyword for System context. [T] Specifies an extended keyword string for the System context variable(s) (variable data).
-kw_prj_extvar[string]	Ext. keyword for external variables. [T] Specifies an extended keyword string for external variables (variable data).

Table 38: Extended keyword project options

Project option, extended keywords	Explanation / [option type]
-c51cs_prj<argument>	C51 constant segment. [E] Default argument: None Specifies where to place C51 constant data. If an extended keyword is specified for a specific type of constant data, the C51 segment will be ignored. 0 (None): No C51 keywords are used. Select this value if the intended target does not support C51 keywords. 1 (CODE segment): Coder-generated constant data are stored in CODE.
-kw_corelogic[string]	Ext. keyword for core model logic. [T] Specifies an extended keyword string for the core model logic struct variable(s) (constant data).
-kw_guardexpr[string]	Ext. keyword for guard expression collection. [T] Specifies an extended keyword string for the guard expression collection variable(s) (constant data).
-kw_actionexpr[string]	Ext. keyword for action expression collection. [T] Specifies an extended keyword string for the action expression collection variable(s) (constant data).
-kw_runtimeinfo[string]	Ext. keyword for runtime info. [T] Specifies an extended keyword string for the runtime info struct variable (constant data). At present, the runtime info struct only contains the digital signature for the VS Project.

Table 38: Extended keyword project options (Continued)

Project option, RealLink	Explanation / [option type]
-kw_rlpd[string]	RealLink protocol data ext. keyword. [T] Specifies an extended keyword string used for RealLink protocol data.
-userlkw[{0 1}]	Use additional RealLink extended keywords. [B] Default argument: 0 Specifies whether to use additional RealLink extended keywords.
-kw_rld[string]	RealLink data ext. keyword. [T] Specifies an extended keyword string used for RealLink symbol table data.
-kw_rlcd[string]	RealLink const data ext. keyword. [T] Specifies an extended keyword string used for RealLink symbol table const data.

Table 39: RealLink project options

Project option, C-SPYLink	Explanation / [option type]
-suppress_cspylink_common_files[<code>{0 1}</code>]	<p>Suppress C-SPYLink common files. [B] Default argument: 0 This option prevents multiple C-SPYLink files from being generated when you are using two or more systems in the same linked image together with the C-SPYLink debug facility.</p>

Table 40: C-SPYLink project options

Project option, API functions	Explanation / [option type]
-seminquiry[<code>{0 1}</code>]	<p>Enable SEM_Inquiry and SEM_GetInput. [B] Default argument: 0 Specifies whether to enable the API functions SEM_Inquiry and SEM_GetInput. If the option is set, the function will be enabled. The option is available if API type is Basic.</p>
-semname[<code>{0 1}</code>]	<p>Enable SEM_Name. [B] Default argument: 0 Specifies whether to enable the API function SEM_Name. If the option is set, the function will be enabled. The option is available if API type is Basic.</p>
-semexpl[<code>{0 1}</code>]	<p>Enable SEM_Expl. [B] Default argument: 0 Specifies whether to enable the API function SEM_Expl. If the option is set, the function will be enabled. The option is available if API type is Basic.</p>
-semstate[<code>{0 1}</code>]	<p>Enable SEM_State. [B] Default argument: 0 Specifies whether to enable the API function SEM_State. If the option is set, the function will be enabled. The option is available if API type is Basic.</p>
-semmachine[<code>{0 1}</code>]	<p>Enable SEM_Machine. [B] Default argument: 0 Specifies whether to enable the API function SEM_Machine. If the option is set, the function will be enabled. The option is available if API type is Basic.</p>
-semforcestate[<code>{0 1}</code>]	<p>Enable SEM_ForceState. [B] Default argument: 0 Specifies whether to enable the API function SEM_ForceState. If the option is set, the function will be enabled. The option is available if API type is Basic.</p>
-semstateall[<code>{0 1}</code>]	<p>Enable SEM_StateAll. [B] Default argument: 0 Specifies whether to enable the API function SEM_StateAll. If the option is set, the function will be enabled. The option is available if API type is Basic.</p>

Table 41: API functions project options

Project option, API functions (Continued)	Explanation / [option type]
<code>-semnextstatechg[{01}]</code>	Enable SEM_NextStateChg. [B] Default argument: 0 Specifies whether to enable the API function SEM_NextStateChg. If the option is set, the function will be enabled. The option is available if API type is Basic.
<code>-semexplabs[{01}]</code>	Enable SEM_ExplAbs. [B] Default argument: 0 Specifies whether to enable the API function SEM_ExplAbs. If the option is set, the function will be enabled. The option is available if API type is Basic.
<code>-semnameabs[{01}]</code>	Enable SEM_NameAbs. [B] Default argument: 0 Specifies whether to enable the API function SEM_NameAbs. If the option is set, the function will be enabled. The option is available if API type is Basic.
<code>-semgetoutputall[{01}]</code>	Enable SEM_GetOutputAll. [B] Default argument: 0 Specifies whether to enable the API function SEM_GetOutputAll. If the option is set, the function will be enabled. The option is available if API type is Basic.
<code>-semgetinputall[{01}]</code>	Enable SEM_GetInputAll. [B] Default argument: 0 Specifies whether to enable the API function SEM_GetInputAll. If the option is set, the function will be enabled. The option is available if API type is Basic.
<code>-semsignalqueueinfo[{01}]</code>	Enable SEM_SignalQueueInfo. [B] Default argument: 0 Specifies whether to enable the API function SEM_SignalQueueInfo. If the option is set, the function will be enabled. The option is available if API type is Basic.

Table 41: API functions project options (Continued)

System options

System option, basic	Explanation / [option type]
-V[string]	System. [T] Specifies a System.

Table 42: Basic system options

System option, file output	Explanation / [option type]
-usepop[{{0 1}}	Use Project output path. [B] Default argument: 1 Specifies whether to use the same output path as the path specified for the Project.
-spath[driveDir]	Output path. [T] Specifies the output path for all generated System files. If the path does not exist, it is created automatically. The path may be a relative path.
-H[pathname]	System header file. [T] Default argument: \$SYS\$.h Specifies the name of the header file containing System level model declarations. If an empty argument is specified for this option, the default name is used.
-S[pathname]	System source file. [T] Default argument: \$SYS\$.c Specifies the name of the source file containing System level model definitions. If an empty argument is specified for this option, the default name is used.
-L[pathname]	System data header file. [T] Default argument: \$SYS\$Data.h Specifies the name of the header file containing additional System level model declarations. If an empty argument is specified for this option, the default name is used.
-K[pathname]	System data source file. [T] Default argument: \$SYS\$Data.c Specifies the name of the source file containing additional System level model definitions. If an empty argument is specified for this option, the default name is used.
-A[pathname]	Action expression header file. [T] Default argument: \$SYS\$Action.h Specifies the name of the header file containing System level action expression declarations. If an empty argument is specified for this option, the default name is used.
-F[pathname]	Action function header file. [T] Specifies the name of the header file containing System level action function declarations. If an empty argument is specified for this option, the declarations are generated in the action expression header file.

Table 43: File output system options

System option, file output (Continued)	Explanation / [option type]
-extvarh[pathname]	External variable header file. [T] Specifies the name of the header file containing System level external variable declarations. If an empty argument is specified for this option, the declarations are generated in the System data header file.
-extvarc[pathname]	External variable source file. [T] Specifies the name of the source file containing System level external variable definitions. If an empty argument is specified for this option, the definitions are generated in the System data source file.
-M[pathname]	Constant header file. [T] Specifies the name of the header file containing System level constant definitions. If an empty argument is specified for this option, the definitions are not generated.
-B[pathname]	System binary file. [T] Default argument: \$SYS\$.sld Specifies the name of the file containing a model of the VS System in a binary loadable format. This file is only used if a System is to be loaded from disk. If an empty argument is specified for this option, the default name is used.

Table 43: File output system options (Continued)

System option, code	Explanation / [option type]
-constcml[{0 1}]	Const core model logic. [B] Default argument: 1 Specifies whether to define the core model logic as a const variable. Only unset this option in exceptional cases. For example, if the target has sufficient and fast RAM, and speed is of highest importance, the option should be unset.
-constguardfpt[{0 1}]	Const guard expression FPT. [B] Default argument: 1 Specifies whether to define the guard expression function pointer table as a const variable. Only unset this option in exceptional cases. For example, if the target has sufficient and fast RAM, and speed is of highest importance, the option should be unset.
-constactionfpt[{0 1}]	Const action expression FPT. [B] Default argument: 1 Specifies whether to define the action expression function pointer table as a const variable. Only unset this option in exceptional cases. For example, if the target has sufficient and fast RAM, and speed is of highest importance, the option should be unset.

Table 44: Code system options

System option, code	Explanation / [option type]
-static[{{0 1}}	<p>Static items. [B]</p> <p>Default argument: 0</p> <p>Specifies whether to define the core model logic, guard expressions and action expressions as static variables and/or functions. Only set this option when using the Basic API file SEMCfgB.c (which is obsolete), and all generated source files are included in this single 'source' file. The option is available if API type is Basic.</p>
-og[{{0 1}}	<p>Merge guard expressions. [B]</p> <p>Default argument: 0</p> <p>Specifies whether to merge guard expressions. If the option is set, speed may be increased since multiple guard expressions associated with a single transition are generated as a compound statement in the generated code. The drawback is that the same guard expression may be generated multiple times if constructs such as entry reactions, exit reactions and/or history states are used. If size is essential, unsetting the option may generate smaller code.</p>
-oa[{{0 1}}	<p>Merge action expressions. [B]</p> <p>Default argument: 0</p> <p>Specifies whether to merge action expressions. If the option is set, speed may be increased since multiple action expressions associated with a single transition are generated as a compound statement in the generated code. The drawback is that the same action expression may be generated multiple times if constructs such as entry reactions, exit reactions and/or history states are used. If size is essential, unsetting the option may generate smaller code.</p>
-noactionfpt[{{0 1}}	<p>Skips generation of action expression collection. [B]</p> <p>Default argument: 0</p> <p>Specifies whether to skip generation of the action expression collection. If the option is set, the macros for executing an action expression (for example SEM_Action/SMP_Action) are of no use, since they depend on the existence of an action expression collection. The option should only be set when generating code compatible with version 3 of visualSTATE.</p>
-omitcontradictiontests[{{0 1}}	<p>Omit contradiction tests. [B]</p> <p>Default argument: 0</p> <p>Turns off the generation of contradiction test code for each transition. Only use this option if you know that your system is free from transition conflicts or if you have particular testing requirements, for example, various branch coverage metrics. Note that if the system is verified in some way to be conflict-free, no test sequence that will exercise the error part of the conflict test can be constructed unless you modify the generated code by inserting test code to manipulate variable values. This option can be used for both human-readable code and table-based code. See also <i>The size of human-readable code</i>, page 253.</p>

Table 44: Code system options (Continued)

System option, readable code	Explanation / [option type]
-splitreadable[<code>{0 1}</code>]	Split readable code. [B] Default argument: 0 This option rewrites a <code><SystemName>VSDeduct()</code> function to use helper functions for event processing. This can be beneficial for very large <code><SystemName>VSDeduct()</code> function, because it reduces the compilation time if aggressive compiler optimization flags are used. It can also overcome arbitrary implementation function size limits of your compiler. This option causes a slight code size and speed overhead.

Table 45: Readable code system options

System option, style	Explanation / [option type]
-struct[<code>identifier</code>]	System structure name. [T] Default argument: System Specifies the name of the core model logic struct variable.
-N[<code>identifier</code>]	Action expression collection name. [T] Default argument: SystemVSAction Specifies the name of the action expression collection.
-lower[<code>{0 1}</code>]	<i>(This option is now ignored)</i> Generates lowercase action function names. [B] Default argument: 0 Specifies whether to generate action function names in lowercase. This option is provided for compatibility with version 3 of visualSTATE.

Table 46: Style system options

System option, ext. keywords	Explanation / [option type]
-c51vs_sys< <code>argument</code> >	C51 variable segment. [E] Default argument: None Specifies where to place C51 variable data. If an extended keyword is specified for a specific type of variable data, the C51 segment will be ignored. 0 (None): No C51 keywords are used. Select this value if the intended target does not support C51 keywords. 1 (DATA segment): Coder-generated variable data are stored in DATA. 2 (IDATA segment): Coder-generated variable data are stored in IDATA. 3 (PDATA segment): Coder-generated variable data are stored in PDATA. 4 (XDATA segment): Coder-generated variable data are stored in XDATA.
-kw_sys_extvar[<code>string</code>]	Ext. keyword for external variables. [T] Specifies an extended keyword string for external variables (variable data).
-kw_intvar[<code>string</code>]	Ext. keyword for internal variables. [T] Specifies an extended keyword string for internal variables (variable data).

Table 47: Extended keywords system options

System option, ext. keywords	Explanation / [option type]
-kw_dbdata[string]	Ext. keyword for double buffer variable. [T] Specifies an extended keyword string for the double buffer variable (variable data).

Table 47: Extended keywords system options (Continued)

System option, names	Explanation / [option type]
-txte<argument>	Event name inclusion. [E] Default argument: No text Specifies the amount of event name inclusion in the core model logic struct. 0 (No text): Includes no element texts in the generated code. 1 (Names included): Includes names in the generated code. If it is necessary to extract names from the VS model running in the target, select this value for the option. See the documentation for the API functions with suffix '_Name' and '_NameAbs'. 2 (Explanations included): Includes explanations in the generated code. If it is necessary to extract explanations from the VS model running in the target, select this value for the option. See the documentation for the API functions with suffix '_Expl' and '_ExplAbs'. 3 (Names and explanations): Includes names and explanations in the generated code.
-sne<argument>	Printing of symbolic event names. [E] Default argument: Do not convert Specifies how to print symbolic event names. 0 (Do not print): Does not generate symbolic element names. 1 (Do not convert): Generates symbolic element names as defined in the VS model. 2 (Convert to uppercase): Generates symbolic element names as defined in the VS model, but converted to uppercase.
-txts<argument>	State name inclusion. [E] Default argument: No text Specifies the amount of state name inclusion in the core model logic struct. 0 (No text): Includes no element texts in the generated code. 1 (Names included): Includes names in the generated code. If it is necessary to extract names from the VS model running in the target, select this value for the option. See the documentation for the API functions with suffix '_Name' and '_NameAbs'. 2 (Explanations included): Includes explanations in the generated code. If it is necessary to extract explanations from the VS model running in the target, select this value for the option. See the documentation for the API functions with suffix '_Expl' and '_ExplAbs'. 3 (Names and explanations): Includes names and explanations in the generated code.

Table 48: Names system options

System option, names (Continued)	Explanation / [option type]
-sns<argument>	Printing of symbolic state names. [E] Default argument: Do not print Specifies how to print symbolic state names. 0 (Do not print): Does not generate symbolic element names. 1 (Do not convert): Generates symbolic element names as defined in the VS model. 2 (Convert to uppercase): Generates symbolic element names as defined in the VS model, but converted to uppercase.
-txta<argument>	Action function name inclusion. [E] Default argument: No text Specifies the amount of action function name inclusion in the core model logic struct. 0 (No text): Includes no element texts in the generated code. 1 (Names included): Includes names in the generated code. If it is necessary to extract names from the VS model running in the target, select this value for the option. See the documentation for the API functions with suffix '_Name' and '_NameAbs'. 2 (Explanations included): Includes explanations in the generated code. If it is necessary to extract explanations from the VS model running in the target, select this value for the option. See the documentation for the API functions with suffix '_Expl' and '_ExplAbs'. 3 (Names and explanations): Includes names and explanations in the generated code.
-sna<argument>	Printing of symbolic action function names. [E] Default argument: Do not print Specifies how to print symbolic action function names. 0 (Do not print): Does not generate symbolic element names. 1 (Do not convert): Generates symbolic element names as defined in the VS model. 2 (Convert to uppercase): Generates symbolic element names as defined in the VS model, but converted to uppercase.
-snm<argument>	Printing of symbolic state machine names. [E] Default argument: Do not print Specifies how to print symbolic state machine names. 0 (Do not print): Does not generate symbolic element names. 1 (Do not convert): Generates symbolic element names as defined in the VS model. 2 (Convert to uppercase): Generates symbolic element names as defined in the VS model, but converted to uppercase.

Table 48: Names system options (Continued)

System option, API functions

-seminitall[*{0|1}*]

Explanation / [option type]

Enable SEM_InitAll or [system_name]SMP_InitAll. [B]

Default argument: 0

Specifies whether to enable the API function SEM_InitAll (Basic API name) or [system_name]SMP_InitAll (Expert API name(s)). If the option is set, the function(s) will be enabled.

Table 49: API functions system options

Docmenter options

This chapter lists the Docmenter options available and how to set them via the command line. You can also set Docmenter options via the Navigator as described in *Setting Verificator, Coder and Docmenter options*, page 29.

Command line syntax

The command line syntax for Docmenter options is:

```
<vsp_filename> [--l] [--@<filename>] <-<option>[argument]>*
```

Specifying `--l` will load options from the specified `vsp` file.

Specifying `--@` will load additional options from the specified file. Each line in the file must contain exactly one option. A line is treated as a comment if the line starts with the character sequence `//`.

Lists of Docmenter options

The individual Docmenter options are listed below. The contents of the lists correspond to the online help of the Docmenter Options dialog box (see *Online help*, page 31).

DOCUMENTER OPTION TYPES

Enumerated options	[E]
Integral options	[I]
Text options	[T]
Boolean options	[B]

<i>Project option, configuration</i>	<i>Explanation / [option type]</i>
-title[string]	Title. [T] Default argument: \$PRJ\$ Specifies the title of the report.
-detail<argument>	Detail level. [E] Default argument: Medium Specifies the detail level of the report. 0 (Low): Explanations, state vectors from Validator test sequence files, and transactions are excluded. 1 (Medium): Explanations and state vectors from Validator test sequence files are excluded. 2 (High): All information related to a Project is included.
-introduction[{{0 1}}]	Include introduction. [B] Default argument: 0 Specifies whether to include an introduction. This section will contain user-written text files.
-design[{{0 1}}]	Include model design. [B] Default argument: 1 Specifies whether to include information on model design. This is the main section of the report. It contains a complete description of the design, including statecharts, transitions, elements, etc.
-test[{{0 1}}]	Include model test. [B] Default argument: 1 Specifies whether to include information on model test. This section contains test files, such as Validator static analysis files, Validator dynamic analysis files, Validator test sequence files and Verificator report files.
-interface[{{0 1}}]	Include model interface. [B] Default argument: 1 Specifies whether to include information on model interface. This section contains a table for each element type that is part of the external interface, that is events, action functions, external variables and constants.
-implementation[{{0 1}}]	Include implementation. [B] Default argument: 1 Specifies whether to include information on implementation. This section contains Coder report files.

Project option, configuration (Continued)**-pseudo_code**[[0|1]]**Explanation / [option type]**

Include pseudo code. [B]

Default argument: 1

Specifies whether to include pseudo code. This section includes pseudo code for the Project.

-element_lists[[0|1]]

Include element lists. [B]

Default argument: 1

Specifies whether to include element lists. This section contains a table for each element type, that is events, event groups, action functions, external variables, internal variables, signals, constants, and external states.

Project option, file input**-usertextfiles**[pathlist]**Explanation / [option type]**

User text files. [T]

Specifies which user text files to include in the report.

-fiCriteria<argument>

File inclusion criteria. [E]

Default argument: Signature and file format match

Specifies the criteria for inclusion of generated files that contain a digital signature such as Validator test sequence files, Coder result files, etc. If an included file does not meet the criteria, either a message, a warning, or an error is generated.

0 (Signature and file format match): The signature (thus also the Project file name) and the file format must all match.

1 (Project file name and format match): The signatures need not match, but the Project file name and format must match.

2 (File format match): The signatures and the Project file name need not match, but the file format must match.

3 (None): No criteria is used to determine which files to include.

-fiLevel<argument>

File inclusion message level. [E]

Default argument: Error

Specifies the message level to use if an included file does not meet the criteria for inclusion of generated files.

0 (Information):

1 (Warning):

2 (Error):

-fiAutoInclude[[0|1]]

Automatically include generated files. [B]

Default argument: 0

Specifies whether to automatically include generated files that contain a digital signature such as Validator test sequence files, Coder result files, etc. By default, the directory searched is the directory where the Project file is located. Only files meeting the file inclusion criteria will be included.

Project option, file input *(Continued)*

	Explanation / [option type]
-fiSearchSubDir[{0 1}]	Auto inclusion searches in subdirectories. [B] Default argument: 1 If generated files are included automatically, setting this option will cause all subdirectories relative to the location of the Project file to be searched.
-vsafiles[pathlist]	Validator static analysis files. [T] Specifies which Validator static analysis files (extension .vsa) to include in the report.
-vdafiles[pathlist]	Validator dynamic analysis files. [T] Specifies which Validator dynamic analysis files (extension .vda) to include in the report.
-vlgfiles[pathlist]	Validator test sequence files. [T] Specifies which Validator test sequence files (extension .vlg) to include in the report.
-vrefiles[pathlist]	Verificator result files. [T] Specifies which Verificator result files (extension .vre) to include in the report.
-crefiles[pathlist]	Coder report files. [T] Specifies which Coder report files (extension .cre) to include in the report.

Project option, file output

	Explanation / [option type]
-of<argument>	Output format. [E] Default argument: RTF Specifies the output format for the report. 0 (RTF): 1 (HTML):
-path[driveDir]	Output path. [T] Default argument: doc\ Specifies the output path for all generated files. If the path does not exist, it is created automatically. The path may be a relative path.
-mf[{0 1}]	Output to multiple files. [B] Default argument: 0 Specifies whether to generate output to multiple files.
-ei[{0 1}]	Embed icons in reports. [B] Default argument: 1 Specifies whether to embed icons (images) within the generated reports.
-ec[{0 1}]	Embed statecharts in reports. [B] Default argument: 1 Specifies whether to embed statecharts (images) within the generated reports.

Project option, file output (Continued)

-pngcharts[[{0|1}]] *Explanation / [option type]*
 Use png files for statecharts (debug option, may throw exception). [B]
 Default argument: 0
 Specifies whether to use png files for statecharts.

Project option, format

-pfe[[{0|1}]] *Explanation / [option type]*
 Parse functional expressions. [B]
 Default argument: 1
 Specifies whether to parse functional expressions. The option should be set in order to generate links from elements used in functional expressions to their respective definitions. The option can be unset when generating documentation for incomplete designs that contain invalid functional expressions.

-lsn[[{0|1}]] *Explanation / [option type]*
 Use long state names. [B]
 Default argument: 0
 Specifies whether to use long state names in state references.

-split[[{0|1}]] *Explanation / [option type]*
 Split transition texts on multiple lines. [B]
 Default argument: 0
 Specifies whether to split transition texts on multiple lines.

-il[[{0|1}]] *Explanation / [option type]*
 Insert links. [B]
 Default argument: 1
 Specifies whether to insert links between uses of elements and their associated definitions.

Project option, page layout

Explanation / [option type]

-top_margin<double>[{"|c|mm|twips|points}] *Explanation / [option type]*
 Top margin. [I]
 Default argument: 2.5 cm
 Specifies the top margin for the report.

-bottom_margin<double>[{"|c|mm|twips|points}] *Explanation / [option type]*
 Bottom margin. [I]
 Default argument: 2.5 cm
 Specifies the bottom margin for the report.

-left_margin<double>[{"|c|mm|twips|points}] *Explanation / [option type]*
 Left margin. [I]
 Default argument: 2.5 cm
 Specifies the left margin for the report.

Project option, page layout (Continued)

-right_margin<double>[{"lcmllmmltwips|points}]

Explanation / [option type]

Right margin. [I]
 Default argument: 2.5 cm
 Specifies the right margin for the report.

-header_from_edge<double>[{"lcmllmmltwips|points}]

Header distance to edge. [I]
 Default argument: 1.25 cm
 Specifies the distance from the top of the page to the header.

-footer_from_edge<double>[{"lcmllmmltwips|points}]

Footer distance to edge. [I]
 Default argument: 1.25 cm
 Specifies the distance from the footer to the bottom of the page.

-paper_type<argument>

Paper type. [E]
 Default argument: A4
 Specifies the paper type. If measurement system is the metric system, default type is A4, otherwise letter.
 0 (User defined):
 1 (Letter):
 2 (Letter Small):
 3 (Tabloid):
 4 (Ledger):
 5 (Legal):
 6 (Statement):
 7 (Executive):
 8 (A3):
 9 (A4):
 10 (A4 Small):
 11 (A5):
 12 (B4 (JIS)):
 13 (B5 (JIS)):
 14 (Folio):
 15 (Quarto):
 16 (10x14):
 17 (11x17):

Project option, page layout (Continued)**Explanation / [option type]**

18 (Note):
 19 (Envelope 9):
 20 (Envelope 10):
 21 (Envelope 11):
 22 (Envelope 12):
 23 (Envelope 14):
 24 (Envelope D1):
 25 (Envelope C5):
 26 (Envelope C3):
 27 (Envelope C4):
 28 (Envelope C6):
 29 (Envelope C65):
 30 (Envelope B4):
 31 (Envelope B5):
 32 (Envelope B6):
 33 (Envelope Italy):
 34 (Envelope Monarch):
 35 (6 3/4 Envelope):
 36 (US Std Fanfold):
 37 (German Std Fanfold):
 38 (German Legal Fanfold):

`-paper_width<double>[{"lcm|mm|twips|points}]`

Paper width. [I]
 Default argument: 0 cm
 Specifies paper width.

`-paper_height<double>[{"lcm|mm|twips|points}]`

Paper height. [I]
 Default argument: 0 cm
 Specifies paper height.

`-paper_orientation<argument>`

Paper orientation. [E]
 Default argument: Portrait
 Specifies paper orientation.
 0 (Portrait):
 1 (Landscape):

Project option, fonts	Explanation / [option type]
-hdr_fname<argument>	Heading font name. [E] Default argument: Arial Specifies the font name used for headings (including text on the front page).
-hdr_fstyle<argument>	Heading font style. [E] Default argument: Bold Specifies the font style used for headings (including text on the front page). 0 (Normal): Normal 1 (Bold): Bold 2 (Italic): Italic 3 (Bold Italic): Bold Italic
-hdr_fsize<integer>	Heading font size. [I] Default argument: 10 Specifies the font size used for headings (including text on the front page).
-code_fname<argument>	Code font name. [E] Default argument: Courier New Specifies the font name used for code (for example pseudo code).
-code_fstyle<argument>	Code font style. [E] Default argument: Normal Specifies the font style used for code (for example pseudo code). 0 (Normal): Normal 1 (Bold): Bold 2 (Italic): Italic 3 (Bold Italic): Bold Italic
-code_fsize<integer>	Code font size. [I] Default argument: 9 Specifies the font size used for code (for example pseudo code).
-text_fname<argument>	Text font name. [E] Default argument: Times New Roman Specifies the font name used for all other texts than headings and code.
-text_fstyle<argument>	Text font style. [E] Default argument: Normal Specifies the font style used for all other texts than headings and code. 0 (Normal): Normal 1 (Bold): Bold 2 (Italic): Italic 3 (Bold Italic): Bold Italic

Project option, fonts (Continued)

-text_fsize<integer> **Explanation / [option type]**
 Text font size. [I]
 Default argument: 10
 Specifies the font size used for all other texts than headings and code.

Project option, front page

-toptext_str[string] **Explanation / [option type]**
 Top text. [T]
 Specifies the top text.

-toptext_justification<argument> **Explanation / [option type]**
 Top text justification. [E]
 Default argument: Centered
 Specifies the justification of the top text.
 0 (Left):
 1 (Right):
 2 (Centered):

-middletext_str[string] **Explanation / [option type]**
 Middle text. [T]
 Default argument: \$PRJ\$
 Specifies the middle text.

-middletext_justification<argument> **Explanation / [option type]**
 Middle text justification. [E]
 Default argument: Centered
 Specifies the justification of the middle text.
 0 (Left):
 1 (Right):
 2 (Centered):

-bottomtext_str[string] **Explanation / [option type]**
 Bottom text. [T]
 Specifies the bottom text.

-bottomtext_justification<argument> **Explanation / [option type]**
 Bottom text justification. [E]
 Default argument: Centered
 Specifies the justification of the bottom text.
 0 (Left):
 1 (Right):
 2 (Centered):

Project option, header/footer

	Explanation / [option type]
-headertextl[string]	Header text left. [T] Specifies the header text aligned left.
-headertextc[string]	Header text centered. [T] Specifies the header text aligned centered.
-headextr[string]	Header text right. [T] Default argument: Page \$PAGES\$ Specifies the header text aligned right.
-header_separator[{{0 1}}]	Separator line after header. [B] Default argument: 1 Specifies whether to include a separator line after the header.
-footertextl[string]	Footer text left. [T] Specifies the footer text aligned left.
-footertextc[string]	Footer text centered. [T] Specifies the footer text aligned centered.
-footextr[string]	Footer text right. [T] Specifies the footer text aligned right.
-footer_separator[{{0 1}}]	Separator line before footer. [B] Default argument: 0 Specifies whether to include a separator line before the footer.

Project option, RTF styles

	Explanation / [option type]
-template[pathname]	Style template. [T] Specifies the style template used by RTF reports.
-ibat[{{0 1}}]	Insert bullet and tab stop in hierarchy. [B] Default argument: 1 Specifies whether to specifically insert a bullet and a tab stop in list hierarchies. The option should be unset when the generated report uses an external template with list styles that by definition include such a list marker and indentation.
-sn_fph[string]	Front page header style name. [T] Default argument: Front Page Header Specifies the name of the front page header style.

Project option, RTF styles (Continued)

	Explanation / [option type]
-sn_fpt[string]	Front page text style name. [T] Default argument: Front Page Text Specifies the name of the front page middle text style.
-sn_fpf[string]	Front page footer style name. [T] Default argument: Front Page Footer Specifies the name of the front page footer style.
-sn_bt[string]	Body text style name. [T] Default argument: Body Text Specifies the name of the body text style.
-sn_rtfcode[string]	Code style name. [T] Default argument: Code Specifies the name of the code style.
-sn_rftoc[string]	TOC heading style name. [T] Default argument: TOC Heading Specifies the name of the heading style of the table of contents.
-sn_hdr[string]	Header style name. [T] Default argument: Header Specifies the name of the header style.
-sn_ftr[string]	Footer style name. [T] Default argument: Footer Specifies the name of the footer style.
-sn_rtfh1[string]	Heading 1 style name. [T] Default argument: Heading 1 Specifies the name of the heading style.
-sn_rtfh2[string]	Heading 2 style name. [T] Default argument: Heading 2 Specifies the name of the heading style.
-sn_rtfh3[string]	Heading 3 style name. [T] Default argument: Heading 3 Specifies the name of the heading style.
-sn_rtfh4[string]	Heading 4 style name. [T] Default argument: Heading 4 Specifies the name of the heading style.
-sn_rtfh5[string]	Heading 5 style name. [T] Default argument: Heading 5 Specifies the name of the heading style.

Project option, RTF styles (Continued)

	Explanation / [option type]
-sn_rtfh6[string]	Heading 6 style name. [T] Default argument: Heading 6 Specifies the name of the heading style.
-sn_rtfh7[string]	Heading 7 style name. [T] Default argument: Heading 7 Specifies the name of the heading style.
-sn_rtfh8[string]	Heading 8 style name. [T] Default argument: Heading 8 Specifies the name of the heading style.
-sn_rtfh9[string]	Heading 9 style name. [T] Default argument: Heading 9 Specifies the name of the heading style.
-sn_lb1[string]	List Bullet 1 style name. [T] Default argument: List Bullet Specifies the name of the list bullet style.
-sn_lb2[string]	List Bullet 2 style name. [T] Default argument: List Bullet 2 Specifies the name of the list bullet style.
-sn_lb3[string]	List Bullet 3 style name. [T] Default argument: List Bullet 3 Specifies the name of the list bullet style.
-sn_lb4[string]	List Bullet 4 style name. [T] Default argument: List Bullet 4 Specifies the name of the list bullet style.
-sn_lb5[string]	List Bullet 5 style name. [T] Default argument: List Bullet 5 Specifies the name of the list bullet style.
-sn_lb6[string]	List Bullet 6 style name. [T] Default argument: List Bullet 6 Specifies the name of the list bullet style.
-sn_lb7[string]	List Bullet 7 style name. [T] Default argument: List Bullet 7 Specifies the name of the list bullet style.
-sn_lb8[string]	List Bullet 8 style name. [T] Default argument: List Bullet 8 Specifies the name of the list bullet style.

Project option, RTF styles (Continued)**-sn_lb9**[string]**Explanation / [option type]**

List Bullet 9 style name. [T]
 Default argument: List Bullet 9
 Specifies the name of the list bullet style.

Project option, HTML styles**-stylesheet**[pathname]**Explanation / [option type]**

Style sheet. [T]
 Specifies the style sheet used by HTML reports.

-html_uhover[[0|1]]

Underline links at mouse over. [B]
 Default argument: 1
 Specifies whether only to underline links when the mouse pointer is over the link.

-html_stl[[0|1]]

Simple table layout. [B]
 Default argument: 1
 Specifies whether to apply a simple layout for tables.

-scn_htmlbody[identifier]

Body style class name. [T]
 Specifies the name of the body style class (HTML element body).

-scn_htmlcode[identifier]

Code style class name. [T]
 Specifies the name of the code style class (HTML element pre).

-scn_htmltoc[identifier]

TOC heading style class name. [T]
 Specifies the name of the heading style class for table of contents (HTML element h1).

-scn_htmlh1[identifier]

Heading 1 style class name. [T]
 Specifies the name of the heading style class (HTML element h1).

-scn_htmlh2[identifier]

Heading 2 style class name. [T]
 Specifies the name of the heading style class (HTML element h2).

-scn_htmlh3[identifier]

Heading 3 style class name. [T]
 Specifies the name of the heading style class (HTML element h3).

-scn_htmlh4[identifier]

Heading 4 style class name. [T]
 Specifies the name of the heading style class (HTML element h4).

-scn_htmlh5[identifier]

Heading 5 style class name. [T]
 Specifies the name of the heading style class (HTML element h5).

-scn_htmlh6[identifier]

Heading 6 style class name. [T]
 Specifies the name of the heading style class (HTML element h6).

-scn_htmlh7[identifier]

Heading 7 style class name. [T]
 Specifies the name of the heading style class (HTML element h7).

Project option, HTML styles *(Continued)*

-scn_htmlh8[identifier]

Explanation / [option type]

Heading 8 style class name. [T]

Specifies the name of the heading style class (HTML element h8).

-scn_htmlh9[identifier]

Heading 9 style class name. [T]

Specifies the name of the heading style class (HTML element h9).

Appendix A: visualSTATE file name extensions

<i>Extension</i>	<i>File type</i>
vsp	visualSTATE Project file
vsr	visualSTATE Statechart file
vnw	visualSTATE workspace file
vtg	visualSTATE Project options file
bk<#>	Designer backup file
vst	Designer interval backup file
vdi	Designer Project diagram information
vdg	Designer Project diagram information (graphical animation)
vsa	Validator static analysis file
vda	Validator dynamic analysis file
vlg	Validator test sequence file
vre	Verificator result file
cre	Coder result file

For detailed information about vsp and vsr files, see *visualSTATE Reference Guide*.

Appendix B: RealLink memory consumption

Using RealLink will increase the size of the generated code. Memory consumption depends on:

- visualSTATE model dependent memory usage
- RealLink API dependent memory usage.

visualSTATE model dependent memory usage

When RealLink is used, the Coder generates additional tables with constant data (CONST DATA) and variable data (DATA). The sizes of these tables largely depend on the visualSTATE application.

The exact memory usage in bytes for CONST DATA memory and DATA can be found by means of the following formulas.

Memory usage in bytes for each visualSTATE Project

$$\text{CONST DATA} = (10 + S) * \text{CDP} + (1 + \text{GEV}) * \text{DP} + 10 * \text{ST} + 13$$

Memory usage in bytes for each visualSTATE System

$$\text{CONST DATA} = 8 * \text{CDP} + \text{FP} + (2 + \text{LEV}) * \text{DP} + (\text{AE} + 1) * \text{AET} + \text{EP} * \text{ST} + (\text{IVT} + 1) * \text{ST}$$

Additional memory usage due to code generation with Expert API

Code generated by the visualSTATE Coder for the visualSTATE Expert API requires additional memory usage which is calculated as follows:

$$\text{DATA} = S * \text{size of SEM_CONTEXT pointer}$$

For all the above formulas the following applies:

S = Number of visualSTATE Systems.

FP = Size of function pointer

CDP = Size of CONST DATA void pointer

DP = Size of DATA pointer

GEV = Number of global external variables

ST = Size of `size_t`

AE = `VS_NOF_ACTION_EXPRESSIONS`

AET = Size of `SEM_ACTION_EXPRESSION_TYPE`

EP = Number of global and local event parameters

IVT = Number of internal variable types used.

Items in monospace font refer to code generated by the visualSTATE Coder.

RealLink API dependent memory usage

The RealLink API memory usage largely depends on the compiler used. *Table 50*, page 410 shows the additional memory consumption by the Basic API when RealLink is used (`RealLink.c` and `RealLink.h`) for an IAR Systems SH7740 32-bit compiler.

Memory	Basic API (all figures in bytes)
CODE	1558
CONST DATA	2
DATA	33
Max. stack used	32

Table 50: RealLink memory consumption, IAR SH7740 32-bit compiler

Appendix C: Source code example

Here you find the source code in Visual Basic for the mobile phone example described in *Interfacing to the Expert DLL using Visual Basic*, page 302.

Mobile phone.frm

```
Public Display_Pict As Byte
Public no_hold_accept As Byte
Public no_hold_down As Byte
Public clr_hold_down As Byte
Public clr_hold_accept As Byte
Public do_time_timer As Byte
Public display_buf As String
Public display_buf_len As Byte
Public last_input As String

Private Sub But_0_Click()
    Dim cc As Byte

    last_input = "KEY_0"
    cc = SEM_QueuePut(1, "KEY_0")
    If cc = SES_QUEUE_OKAY Then
        Call DispatchOutput
    Else
        Call SEM_VBErrorHandler("SEM_QueuePut", cc)
    End If
End Sub

Private Sub But_0_DblClick()
    Dim cc As Byte

    cc = SEM_QueuePut(1, "KEY_0")
    If cc = SES_QUEUE_OKAY Then
        Call DispatchOutput
    Else
        Call SEM_VBErrorHandler("SEM_QueuePut", cc)
    End If
End Sub
```

```
End Sub
```

```
Private Sub But_1_Click()  
    Dim cc As Byte  
  
    last_input = "KEY_1"  
    cc = SEM_QueuePut(1, "KEY_1")  
    If cc = SES_QUEUE_OKAY Then  
        Call DispatchOutput  
    Else  
        Call SEM_VBErrorHandler("SEM_QueuePut", cc)  
    End If  
End Sub
```

```
Private Sub But_1_DblClick()  
    Dim cc As Byte  
  
    cc = SEM_QueuePut(1, "KEY_1")  
    If cc = SES_QUEUE_OKAY Then  
        Call DispatchOutput  
    Else  
        Call SEM_VBErrorHandler("SEM_QueuePut", cc)  
    End If  
End Sub
```

```
Private Sub But_2_Click()  
    Dim cc As Byte  
  
    last_input = "KEY_2"  
    cc = SEM_QueuePut(1, "KEY_2")  
    If cc = SES_QUEUE_OKAY Then  
        Call DispatchOutput  
    Else  
        Call SEM_VBErrorHandler("SEM_QueuePut", cc)  
    End If  
End Sub
```

```
Private Sub But_2_DblClick()  
    Dim cc As Byte  
  
    cc = SEM_QueuePut(1, "KEY_2")  
    If cc = SES_QUEUE_OKAY Then  
        Call DispatchOutput
```



```
Else
    Call SEM_VBErrorHandler("SEM_QueuePut", cc)
End If

End Sub

Private Sub But_3_Click()
    Dim cc As Byte

    last_input = "KEY_3"
    cc = SEM_QueuePut(1, "KEY_3")
    If cc = SES_QUEUE_OKAY Then
        Call DispatchOutput
    Else
        Call SEM_VBErrorHandler("SEM_QueuePut", cc)
    End If
End Sub

Private Sub But_3_DblClick()
    Dim cc As Byte

    cc = SEM_QueuePut(1, "KEY_3")
    If cc = SES_QUEUE_OKAY Then
        Call DispatchOutput
    Else
        Call SEM_VBErrorHandler("SEM_QueuePut", cc)
    End If

End Sub

Private Sub But_4_Click()
    Dim cc As Byte

    last_input = "KEY_4"
    cc = SEM_QueuePut(1, "KEY_4")
    If cc = SES_QUEUE_OKAY Then
        Call DispatchOutput
    Else
        Call SEM_VBErrorHandler("SEM_QueuePut", cc)
    End If
End Sub

Private Sub But_4_DblClick()
```

```

Dim cc As Byte

cc = SEM_QueuePut(1, "KEY_4")
If cc = SES_QUEUE_OKAY Then
    Call DispatchOutput
Else
    Call SEM_VBErrorHandler("SEM_QueuePut", cc)
End If

End Sub

Private Sub But_5_Click()
Dim cc As Byte

last_input = "KEY_5"
cc = SEM_QueuePut(1, "KEY_5")
If cc = SES_QUEUE_OKAY Then
    Call DispatchOutput
Else
    Call SEM_VBErrorHandler("SEM_QueuePut", cc)
End If
End Sub

Private Sub But_5_DblClick()
Dim cc As Byte

cc = SEM_QueuePut(1, "KEY_5")
If cc = SES_QUEUE_OKAY Then
    Call DispatchOutput
Else
    Call SEM_VBErrorHandler("SEM_QueuePut", cc)
End If

End Sub

Private Sub But_6_Click()
Dim cc As Byte

last_input = "KEY_6"
cc = SEM_QueuePut(1, "KEY_6")
If cc = SES_QUEUE_OKAY Then
    Call DispatchOutput
Else
    Call SEM_VBErrorHandler("SEM_QueuePut", cc)

```

```
End If
End Sub

Private Sub But_6_DblClick()
    Dim cc As Byte

    cc = SEM_QueuePut(1, "KEY_6")
    If cc = SES_QUEUE_OKAY Then
        Call DispatchOutput
    Else
        Call SEM_VBErrorHandler("SEM_QueuePut", cc)
    End If
End Sub

Private Sub But_7_Click()
    Dim cc As Byte

    last_input = "KEY_7"
    cc = SEM_QueuePut(1, "KEY_7")
    If cc = SES_QUEUE_OKAY Then
        Call DispatchOutput
    Else
        Call SEM_VBErrorHandler("SEM_QueuePut", cc)
    End If
End Sub

Private Sub But_7_DblClick()
    Dim cc As Byte

    cc = SEM_QueuePut(1, "KEY_7")
    If cc = SES_QUEUE_OKAY Then
        Call DispatchOutput
    Else
        Call SEM_VBErrorHandler("SEM_QueuePut", cc)
    End If
End Sub

Private Sub But_8_Click()
    Dim cc As Byte

    last_input = "KEY_8"
```

```

cc = SEM_QueuePut(1, "KEY_8")
If cc = SES_QUEUE_OKAY Then
    Call DispatchOutput
Else
    Call SEM_VBErrorHandler("SEM_QueuePut", cc)
End If
End Sub

```

```

Private Sub But_8_DblClick()
    Dim cc As Byte

    cc = SEM_QueuePut(1, "KEY_8")
    If cc = SES_QUEUE_OKAY Then
        Call DispatchOutput
    Else
        Call SEM_VBErrorHandler("SEM_QueuePut", cc)
    End If

End Sub

```

```

Private Sub But_9_Click()
    Dim cc As Byte

    last_input = "KEY_9"
    cc = SEM_QueuePut(1, "KEY_9")
    If cc = SES_QUEUE_OKAY Then
        Call DispatchOutput
    Else
        Call SEM_VBErrorHandler("SEM_QueuePut", cc)
    End If

End Sub

```

```

Private Sub But_9_DblClick()
    Dim cc As Byte

    cc = SEM_QueuePut(1, "KEY_9")
    If cc = SES_QUEUE_OKAY Then
        Call DispatchOutput
    Else
        Call SEM_VBErrorHandler("SEM_QueuePut", cc)
    End If

End Sub

```

```

Private Sub But_CLR_Click()
    Dim cc As Byte

    If clr_hold_accept = False Then
        last_input = "KEY_CLR"
        cc = SEM_QueuePut(1, "KEY_CLR")
        If cc = SES_QUEUE_OKAY Then
            Call DispatchOutput
        Else
            Call SEM_VBErrorHandler("SEM_QueuePut", cc)
        End If
        Form1.Timer3.Enabled = False
    Else
        clr_hold_accept = False
        last_input = KEY_CLR_HOLD
    End If
End Sub

Private Sub But_CLR_DblClick()
    Dim cc As Byte

    If Key_Clr_Hold_Ok = False Then
        cc = SEM_QueuePut(1, "KEY_CLR")
        If cc = SES_QUEUE_OKAY Then
            Call DispatchOutput
        Else
            Call SEM_VBErrorHandler("SEM_QueuePut", cc)
        End If
    End If
End Sub

Private Sub But_CLR_MouseDown(Button As Integer, Shift As Integer,
    x As
    Single, y As Single)
    clr_hold_down = True
    Form1.Timer3.Enabled = True
End Sub

Private Sub But_CLR_MouseUp(Button As Integer, Shift As Integer, x
    As
    Single, y As Single)
    clr_hold_down = False
End Sub

```

```

Private Sub But_No_Click()
    Dim cc As Byte

    If no_hold_accept = False Then
        last_input = "KEY_NO"
        cc = SEM_QueuePut(1, "KEY_NO")
        If cc = SES_QUEUE_OKAY Then
            Call DispatchOutput
        Else
            Call SEM_VBErrorHandler("SEM_QueuePut", cc)
        End If
        Form1.Timer1.Enabled = False
    Else
        no_hold_accept = False
        last_input = "KEY_NO_HOLD"
    End If
End Sub

Private Sub But_No_MouseDown(Button As Integer, Shift As Integer,
    x As
    Single, y As Single)
    no_hold_down = True
    Form1.Timer1.Enabled = True
End Sub

Private Sub But_No_MouseUp(Button As Integer, Shift As Integer, x
    As
    Single, y As Single)
    no_hold_down = False
End Sub

Private Sub But_Square_Click()
    Dim cc As Byte

    last_input = "KEY_NUMBER"
    cc = SEM_QueuePut(1, "KEY_NUMBER")
    If cc = SES_QUEUE_OKAY Then
        Call DispatchOutput
    Else
        Call SEM_VBErrorHandler("SEM_QueuePut", cc)
    End If
End Sub

```

```
Private Sub But_Square_DblClick()  
    Dim cc As Byte  
  
    cc = SEM_QueuePut(1, "KEY_NUMBER")  
    If cc = SES_QUEUE_OKAY Then  
        Call DispatchOutput  
    Else  
        Call SEM_VBErrorHandler("SEM_QueuePut", cc)  
    End If  
End Sub  
  
Private Sub But_Star_Click()  
    Dim cc As Byte  
  
    last_input = "KEY_ASTERIX"  
    cc = SEM_QueuePut(1, "KEY_ASTERIX")  
    If cc = SES_QUEUE_OKAY Then  
        Call DispatchOutput  
    Else  
        Call SEM_VBErrorHandler("SEM_QueuePut", cc)  
    End If  
End Sub  
  
Private Sub But_Star_DblClick()  
    Dim cc As Byte  
  
    cc = SEM_QueuePut(1, "KEY_ASTERIX")  
    If cc = SES_QUEUE_OKAY Then  
        Call DispatchOutput  
    Else  
        Call SEM_VBErrorHandler("SEM_QueuePut", cc)  
    End If  
End Sub  
  
Private Sub But_Yes_Click()  
    Dim cc As Byte  
  
    last_input = "KEY_YES"  
    cc = SEM_QueuePut(1, "KEY_YES")  
    If cc = SES_QUEUE_OKAY Then  
        Call DispatchOutput  
    Else
```

```

        Call SEM_VBErrorHandler("SEM_QueuePut", cc)
    End If
End Sub

Private Sub Command1_Click()
    Dim cc As Byte

    cc = SEM_QueuePut(1, "SE_RESET")
    If cc = SES_QUEUE_OKAY Then
        Call DispatchOutput
    Else
        Call SEM_VBErrorHandler("SEM_QueuePut", cc)
    End If
End Sub

Private Sub Command2_Click()
    Unload Me
End Sub

Private Sub Form_Load()
    Dim cc As Byte
    Dim hInst As Integer

    Call RepositionImages

    Form1.Visible = True

    cc = SEM_Load(0) 'Dll loads MOBILE.SLD
    If cc = SES_OKAY Then
        Call SEM_Init
        cc = SEM_GetInitCC()
        If cc <> SES_OKAY Then
            Call SEM_VBErrorHandler("SEM_Init", cc)
        End If
        Call SEM_QueueInit
        cc = SEM_QueueCreate(1, 1, 4)
        If cc <> SES_QUEUE_OKAY Then
            Call SEM_VBErrorHandler("SEM_QueueCreate", cc)
        End If
        cc = SEM_QueueCreate(2, 10, 4)
        If cc <> SES_QUEUE_OKAY Then
            Call SEM_VBErrorHandler("SEM_QueueCreate", cc)
        End If
    Else

```



```

        Call SEM_VBErrorHandler("SEM_Load", cc)
        MsgBox "Program terminated..."
    End
End If
cc = SEM_QueuePut(1, "SE_RESET")
If cc = SES_QUEUE_OKAY Then
    Call DispatchOutput
Else
    Call SEM_VBErrorHandler("SEM_QueuePut", cc)
End If
End Sub

Private Sub Form_Terminate()
    Call SEM_Free
    SEM_QueueDestroy (1)
    SEM_QueueDestroy (2)
End Sub

Private Sub List3_DblClick()
    Dim cc As Byte

    last_input = List3.Text
    cc = SEM_QueuePut(1, List3.Text)
    If cc = SES_QUEUE_OKAY Then
        Call DispatchOutput
    Else
        Call SEM_VBErrorHandler("SEM_QueuePut", cc)
    End If
End Sub

Private Sub Timer1_Timer()
    Dim cc As Byte

    If no_hold_down = True Then
        cc = SEM_QueuePut(1, "KEY_NO_HOLD")
        If cc = SES_QUEUE_OKAY Then
            Call DispatchOutput
        Else
            Call SEM_VBErrorHandler("SEM_QueuePut", cc)
        End If
        Timer1.Enabled = False
        no_hold_accept = True
    End If
End Sub

```

```
End Sub
```

```
Private Sub Timer2_Timer()
If do_time_timer = True Then
    Label3.Caption = Format(Time, "hh:mm") & "  "
End If
End Sub
```

```
Private Sub Timer3_Timer()
    Dim cc As Byte

    If clr_hold_down Then
        clr_hold_accept = True
        Timer3.Enabled = False
        cc = SEM_QueuePut(1, "KEY_CLR_HOLD")
        If cc = SES_QUEUE_OKAY Then
            Call DispatchOutput
        Else
            Call SEM_VBErrorHandler("SEM_QueuePut", cc)
        End If
    End If
End Sub
```

```
Private Sub Timer4_Timer()
    Timer4 = False
    do_time_timer = True
    Label1.Caption = "WORLD  "
    Label3.Caption = Format(Time, "hh:mm") & "  "
End Sub
```

```
Private Sub Timer5_Timer()
    Dim do_beep As Byte

    do_beep = False
    Select Case Display_Pict
        Case 10
            Display_Pict = 13
            do_beep = True
        Case 11
            Display_Pict = 14
            do_beep = True
        Case 12
            Display_Pict = 15
```

```

        do_beep = True
    Case 13
        Display_Pict = 10
    Case 14
        Display_Pict = 11
    Case 15
        Display_Pict = 12
End Select
If Display_Pict <> 0 Then
    Form1.Image1.Picture = LoadResPicture(Display_Pict, 0)
    If Display_Pict>12 Then
        Form1.Image2.Picture = LoadResPicture(20, 0)
    Else
        Form1.Image2.Picture = LoadPicture()
    End If
End If
If do_beep Then
    For i = 1 To 2
        Beep
    Next i
End If
End Sub

```

Main.bas

```

Public Sub DispatchOutput()
    Dim cc1 As Byte
    Dim cc2 As Byte
    Dim event As Integer
    Dim iptr As Integer
    Dim str As String * 129
    Dim trimstr As String
    Dim strlen As Byte
    Static Busy As Boolean

    If Busy <> True Then
        Busy = True

    Do While SEM_QueueAllGet(event) = SES_QUEUE_OKAY
        'If cc1 <> SES_QUEUE_OKAY Then
        ' Exit Do

        'Common initialization field performed between every incoming
        'event
        Form1.List1.Clear
    
```

```

Form1.List2.Clear
Form1.List3.Clear

cc2 = SEM_Deduct(event)
If cc2 <> SES_OKAY Then
    Call SEM_VBErrorHandler("SEM_Deduct", cc2)
End If

Do
    cc2 = SEM_GetOutput(iptr)
    If cc2 = SES_FOUND Then

        If SEM_Name(OUTPUTTYPE, iptr, str, 128) = SES_OKAY Then
            Call RemoveAsciiZeroAndTrim(str, trimstr)

            Select Case trimstr
                Case "CLEAR_DISP"
                    Form1.Label1.Caption = ""
                    Form1.Label2.Caption = ""
                    Form1.Label3.Caption = ""
                    Form1.display_buf = ""
                    Form1.display_buf_len = 0
                Case "DELETE_DIG"
                    Dim cc As Byte

                    Form1.List1.AddItem "DELETE_DIG"
                    If Form1.display_buf_len < 1 Then 'If buffer is
                        'empty
                        Form1.display_buf = ""
                        Form1.display_buf_len = 0
                    Else
                        If Form1.display_buf_len = 1 Then 'If last digit
                            'change state to stand_by
                            cc = SEM_QueuePut(1, "INTERN_CLR")
                            If cc = SES_QUEUE_OKAY Then
                                Call DispatchOutput
                            Else
                                Call SEM_VBErrorHandler("SEM_QueuePut", cc)
                            End If
                        Else 'Remove last digit from display
                            Form1.display_buf_len =
                                Form1.display_buf_len - 1
                            Form1.display_buf = Left(Form1.display_buf,
                                Form1.display_buf_len)
                        End If
                    End If
                End Select
            End If
        End If
    End Do

```

```

End If
Form1.Label3.Caption = Form1.display_buf
Case "DISCONNECTED"
Form1.List1.AddItem "DISCONNECTED"
Form1.Timer4.Enabled = True
Case "DISPLAY_CON"
Form1.Timer4.Enabled = False
Form1.List1.AddItem "DISPLAY_CON"
Form1.Label11.Caption = "CONNECTED"
Case "DISPLAY_MAX_SIG"
Form1.List1.AddItem "DISPLAY_MAX_SIG"
Form1.Display_Pict = 10
Form1.Imagel.Picture =
LoadResPicture(Form1.Display_Pict, 0)
Case "DISPLAY_MIN_SIG"
Form1.List1.AddItem "DISPLAY_MIN_SIG"
Form1.Display_Pict = 11
Form1.Imagel.Picture =
LoadResPicture(Form1.Display_Pict, 0)
Case "DISPLAY_NO_SIG"
Form1.List1.AddItem "DISPLAY_NO_SIG"
Form1.Timer5.Enabled = False
Form1.Display_Pict = 12
Form1.Imagel.Picture =
LoadResPicture(Form1.Display_Pict, 0)
Form1.Image2.Picture = LoadPicture()
Form1.Timer4.Enabled = True
Case "FIXGS"
Form1.List1.AddItem "FIXGS"
Case "INITIALIZE" 'OK
Form1.List1.AddItem "INITIALIZE"
Form1.Timer1.Enabled = False
Form1.Timer2.Enabled = True
Form1.Timer3.Enabled = False
Form1.Timer4.Enabled = False
Form1.Timer5.Enabled = False
Form1.Display_Pict = 0
Form1.do_time_timer = False
Form1.no_hold_accept = False
Form1.no_hold_down = False
Form1.Picture = LoadResPicture(1, 0)
Form1.Imagel.Picture = LoadPicture()
Form1.Image2.Picture = LoadPicture()
Form1.Label11.Caption = ""
Form1.Label2.Caption = ""

```

```

Form1.Label3.Caption = ""
Form1.display_buf = ""
Form1.display_buf_len = 0
Case "INT_RESTORE" 'OK
Form1.List1.AddItem "INT_RESTORE"
Form1.Timer4.Enabled = True
Case "LIGHT_BLINK" 'OK
Form1.List1.AddItem "LIGHT_BLINK"
Form1.do_time_timer = False
Form1.Label11.Caption = "CALLING "
Case "LIGHT_OFF" 'OK
Form1.List1.AddItem "LIGHT_OFF"
Form1.Picture = LoadResPicture(1, 0)
Form1.Image1.Picture = LoadPicture()
Form1.Image2.Picture = LoadPicture()
Case "LIGHT_ON" 'OK
Form1.List1.AddItem "LIGHT_ON"
If Form1.Display_Pict = 13 Then
Form1.Display_Pict = 10
Else
If Form1.Display_Pict = 14 Then
Form1.Display_Pict = 11
Else
If Form1.Display_Pict = 15 Then
Form1.Display_Pict = 12
End If
End If
End If
Form1.Image1.Picture =
LoadResPicture(Form1.Display_Pict, 0)
Form1.Image2.Picture = LoadPicture()
Form1.Timer4.Enabled = True
Case "OPENDISPLAY" 'OK
Form1.List1.AddItem "OPENDISPLAY"
Form1.Picture = LoadResPicture(2, 0)
Form1.Display_Pict = 10
Beep
Case "OPENPORT1"
Form1.List1.AddItem "OPENPORT1"
Case "RESTORE_DISPLAY" 'OK
Form1.List1.AddItem "RESTORE_DISPLAY"
Form1.Timer4.Enabled = True
Form1.Display_Pict = 10
Case "RING_OFF"
Form1.List1.AddItem "RING_OFF"

```

```

Form1.Timer5.Enabled = False
Case "RING_ON"
    Form1.List1.AddItem "RING_ON"
    Form1.Timer5.Enabled = True
Case "SENDNUMBER"
    Form1.List1.AddItem "SENDNUMBER"
    Form1.Label1.Caption = "CALLING "
Case "STORE_DIG" 'OK
Form1.do_time_timer = False
Form1.List1.AddItem "STORE_DIG"
Select Case Form1.last_input
    Case "KEY_0"
        Form1.display_buf = Form1.display_buf & "0"
        Form1.display_buf_len =
            Form1.display_buf_len + 1
    Case "KEY_1"
        Form1.display_buf = Form1.display_buf & "1"
        Form1.display_buf_len =
            Form1.display_buf_len + 1
    Case "KEY_2"
        Form1.display_buf = Form1.display_buf & "2"
        Form1.display_buf_len =
            Form1.display_buf_len + 1
    Case "KEY_3"
        Form1.display_buf = Form1.display_buf & "3"
        Form1.display_buf_len =
            Form1.display_buf_len + 1
    Case "KEY_4"
        Form1.display_buf = Form1.display_buf & "4"
        Form1.display_buf_len =
            Form1.display_buf_len + 1
    Case "KEY_5"
        Form1.display_buf = Form1.display_buf & "5"
        Form1.display_buf_len =
            Form1.display_buf_len + 1
    Case "KEY_6"
        Form1.display_buf = Form1.display_buf & "6"
        Form1.display_buf_len =
            Form1.display_buf_len + 1
    Case "KEY_7"
        Form1.display_buf = Form1.display_buf & "7"
        Form1.display_buf_len =
            Form1.display_buf_len + 1
    Case "KEY_8"
        Form1.display_buf = Form1.display_buf & "8"

```

```

        Form1.display_buf_len =
            Form1.display_buf_len + 1
    Case "KEY_9"
        Form1.display_buf = Form1.display_buf & "9"
        Form1.display_buf_len =
            Form1.display_buf_len + 1
    Case "KEY_NUMBER"
        Form1.display_buf = Form1.display_buf & "#"
        Form1.display_buf_len =
            Form1.display_buf_len + 1
    Case "KEY_ASTERIX"
        Form1.display_buf = Form1.display_buf & "*"
        Form1.display_buf_len =
            Form1.display_buf_len + 1
    End Select
Case "SWITCHGS"
    Form1.List1.AddItem "SWITCHGS"
Case "SWITCH_OFF"
    Form1.List1.AddItem "SWITCH_OFF"
    Form1.do_time_timer = False
    Beep
Case "TESTGS"
    Form1.List1.AddItem "TESTGS"
Case "UPDATE_DISP" 'OK
    Form1.Label1.Caption = ""
    Form1.Label3.Caption = Form1.display_buf
Case Else
    MsgBox "Output Var." & "" & trimstr &
        "" & "is not Defined"
    End Select
End If
End If
Loop Until cc2 <> SES_FOUND

If cc2 <> SES_OKAY Then
    Call SEM_VBErrorHandler("SEM_GetOutput", cc2)
End If

cc2 = SEM_NextState()
If cc2 <> SES_OKAY Then
    Call SEM_VBErrorHandler("SEM_NextState", cc2)
End If

Call UpdateStateListBox

```



```

cc2 = SEM_Inquiry()
If cc2 <> SES_OKAY Then
    Call SEM_VBErrorHandler("SEM_Inquiry", cc2)
End If
Do
    cc2 = SEM_GetInput(iptr, 0)
    If cc2 = SES_FOUND Then
        If SEM_Name(INPUTTYPE, iptr, str, 128) = SES_OKAY Then
            Call RemoveAsciiZeroAndTrim(str, trimstr)
            Form1.List3.AddItem trimstr
        End If
    End If
    Loop Until cc2 <> SES_FOUND

    If cc2 <> SES_OKAY Then
        Call SEM_VBErrorHandler("SEM_GetInput", cc2)
    End If

Loop
Busy = False
End If
End Sub

```

Utility.bas

```

Public Sub RepositionImages()
Dim x As Integer
Dim y As Integer
Dim w As Integer
Dim h As Integer

x = Screen.TwipsPerPixelX
y = Screen.TwipsPerPixelY
w = Screen.Width / x
h = Screen.Height / y

'Form1
Form1.Top = y * ((h - 577) / 2)
If Form1.Top < 0 Then
    Form1.Top = 0
End If
Form1.Left = x * ((w - 396) / 2)
Form1.Width = x * 396
Form1.Height = y * 577

```

```
'Input text
Form1.Label4.Top = y * 0
Form1.Label4.Left = x * 184
Form1.Label4.Width = x * 200
Form1.Label4.Height = y * 20

'Input list box
Form1.List3.Top = y * 21
Form1.List3.Left = x * 184
Form1.List3.Width = x * 200
Form1.List3.Height = y * 262

'State text
Form1.Label5.Top = y * 290
Form1.Label5.Left = x * 184
Form1.Label5.Width = x * 200
Form1.Label5.Height = y * 20

'State list box
Form1.List2.Top = y * 310
Form1.List2.Left = x * 184
Form1.List2.Width = x * 200
Form1.List2.Height = y * 80

'Output text
Form1.Label6.Top = y * 398
Form1.Label6.Left = x * 184
Form1.Label6.Width = x * 200
Form1.Label6.Height = y * 20

'Output list box
Form1.List1.Top = y * 418
Form1.List1.Left = x * 184
Form1.List1.Width = x * 200
Form1.List1.Height = y * 80

'SE_RESET
Form1.Command1.Top = y * 510
Form1.Command1.Left = x * 184
Form1.Command1.Width = x * 100
Form1.Command1.Height = y * 28

'Exit
Form1.Command2.Top = y * 510
Form1.Command2.Left = x * 285
Form1.Command2.Width = x * 100
Form1.Command2.Height = y * 28
```

```
'Diode
Form1.Image2.Top = y * 124
Form1.Image2.Left = x * 110
Form1.Image2.Width = x * 8
Form1.Image2.Height = y * 10

'Display
Form1.Image1.Top = y * 250
Form1.Image1.Left = x * 32
Form1.Image1.Width = x * 123
Form1.Image1.Height = y * 44

'Top display text line
Form1.Label1.Top = y * 254
Form1.Label1.Left = x * 47
Form1.Label1.Width = x * 91
Form1.Label1.Height = y * 13

'Middle display text line
Form1.Label2.Top = y * 266
Form1.Label2.Left = x * 47
Form1.Label2.Width = x * 91
Form1.Label2.Height = y * 13

'Height display text line
Form1.Label3.Top = y * 278
Form1.Label3.Left = x * 38
Form1.Label3.Width = x * 110
Form1.Label3.Height = y * 13

'Key_Yes
Form1.But_Yes.Top = y * 312
Form1.But_Yes.Left = x * 33
Form1.But_Yes.Width = x * 51
Form1.But_Yes.Height = y * 30

'Key_No
Form1.But_No.Top = y * 312
Form1.But_No.Left = x * 102
Form1.But_No.Width = x * 50
Form1.But_No.Height = y * 30

'Key_Clr
Form1.But_CLR.Top = y * 350
Form1.But_CLR.Left = x * 75
Form1.But_CLR.Width = x * 34
```

```
Form1.But_CLR.Height = y * 24

'Key_1
Form1.But_1.Top = y * 381
Form1.But_1.Left = x * 30
Form1.But_1.Width = x * 34
Form1.But_1.Height = y * 24

'Key_2
Form1.But_2.Top = y * 381
Form1.But_2.Left = x * 75
Form1.But_2.Width = x * 34
Form1.But_2.Height = y * 24

'Key_3
Form1.But_3.Top = y * 381
Form1.But_3.Left = x * 120
Form1.But_3.Width = x * 34
Form1.But_3.Height = y * 24

'Key_4
Form1.But_4.Top = y * 413
Form1.But_4.Left = x * 30
Form1.But_4.Width = x * 34
Form1.But_4.Height = y * 24

'Key_5
Form1.But_5.Top = y * 413
Form1.But_5.Left = x * 75
Form1.But_5.Width = x * 108
Form1.But_5.Height = y * 24

'Key_6
Form1.But_6.Top = y * 413
Form1.But_6.Left = x * 120
Form1.But_6.Width = x * 34
Form1.But_6.Height = y * 24

'Key_7
Form1.But_7.Top = y * 445
Form1.But_7.Left = x * 30
Form1.But_7.Width = x * 34
Form1.But_7.Height = y * 24

'Key_8
Form1.But_8.Top = y * 445
Form1.But_8.Left = x * 75
```

```

Form1.But_8.Width = x * 34
Form1.But_8.Height = y * 24

'Key_9
Form1.But_9.Top = y * 445
Form1.But_9.Left = x * 120
Form1.But_9.Width = x * 34
Form1.But_9.Height = y * 24

'Key_0
Form1.But_0.Top = y * 477
Form1.But_0.Left = x * 75
Form1.But_0.Width = x * 34
Form1.But_0.Height = y * 24

'Key_Star
Form1.But_Star.Top = y * 477
Form1.But_Star.Left = x * 30
Form1.But_Star.Width = x * 34
Form1.But_Star.Height = y * 24

'Key_Number
Form1.But_Square.Top = y * 477
Form1.But_Square.Left = x * 120
Form1.But_Square.Width = x * 34
Form1.But_Square.Height = y * 24

End Sub
Public Sub UpdateStateListBox()
    Dim cc As Byte
    Dim is_on As Byte
    Dim iState As Integer
    Dim str As String * 129
    Dim trimstr As String
    Dim Machine As Integer

    Form1.List2.Clear
    is_on = True
    For Machine = 0 To SEM_NoMachines - 1 Step 1
        cc = SEM_State(Machine, iState)
        If cc <> SES_FOUND Then
            Call SEM_VBErrorHandler("SEm_State", cc)
            Exit For
        End If
        cc = SEM_Name(STATETYPE, iState, str, 128)
        If cc <> SES_OKAY Then

```

```
        Call SEM_VBErrorHandler("SEM_Name", cc)
        Exit For
    End If
    Call RemoveAsciiZeroAndTrim(str, trimstr)
    If trimstr = "power_off" Then
        is_on = False
    End If
    Form1.List2.AddItem trimstr
Next Machine
End Sub
```

```
Public Sub RemoveAsciiZeroAndTrim(
    ByVal sText1 As String, ByRef sText2
As String)
    Dim sSearch As String
    Dim iPos As Integer
    Dim iLength As Integer

    If Len(sText1)>0 Then
        sSearch = String(1, 0)
        Trim (sText1)
        iPos = InStr(sText1, sSearch)
        sText2 = Mid$(sText1, 1, iPos - 1)
    Else
        sText2 = ""
    End If
End Sub
```

Appendix D: Handling visualSTATE files from previous versions

It is possible to use visualSTATE models in version 6 that were created with visualSTATE version 4.x.

In order to be able to modify the models with version 6 programs, the models must be converted from the previous version 4.x file format to version 6 file format. This is done with the conversion facility *ConvertF1ToF2.exe* which is included in the visualSTATE software.

The models can be converted in two ways: manually, as described here, or by using the version 6 Navigator or Designer (see *Handling Projects from previous visualSTATE versions*, page 35).

In the following, the file format used in visualSTATE version 4.x will be called *format 1*.

Manual conversion from format 1 to 6 format

visualSTATE models saved in file format 1 can be converted to visualSTATE version 6 by using the program *ConvertF1ToF2.exe*. The program will convert the specified Project with all its files to a new Project file and related `vsr` files.

The conversion program must be run from a DOS command prompt. The program has the following call syntax:

```
<Project> <Output directory>
```

where

<Project> is the name of the source visualSTATE Project.

<Output directory> is the directory where the the new converted files should be placed.

Example

ConvertF1ToF2 project.vsp converted

The format 1 Project `project.vsp` will be converted to a new Project with the same name and placed in the output directory `converted`. All files in the Project will also be converted and placed in the directory `converted`.

All files of the previous visualSTATE version will be copied to an automatically created directory named `Backup` below the Project directory.

A

- accessing files under source code control 38
- action function editor 93
- action function parameters, arguments for 91
- action function return values, setting. 177
- action functions
 - activated. 126
 - Altia. 283
 - creating 91
 - external declarations 246
 - in external C files. 93
 - simulating with Altia 285
 - unbound, Altia 283
 - unused 124
- action processing 8
- action sequences 8–9
- action side. 83
- Action window 154
- actions
 - produced by sent events 165
 - return values. 177
 - setting up breakpoints for. 174
- activated action functions 126
- activated assignments 126
- activated guard expressions. 125
- activating custom commands 44
- activating events 304
- activating instances 178
- activating the Verificator 137, 141
- activation of elements 125
- active events 154, 164
- adding files to source code control 37
- alias, states 68
- aligning objects in statechart diagrams 63
- Altia 279
 - action functions 283
 - animation of objects. 286
 - binding external signals 280
 - binding visualSTATE elements to objects 284
 - button objects. 280
 - command line parameters 289
 - communication link 279
 - connecting external signals 280–281
 - connecting visualSTATE elements to objects 283
 - connectors 280, 284, 286
 - creating new designs 283
 - defining properties 289
 - disconnecting objects and elements 285
 - Edit mode. 285
 - editing designs 283
 - establishing connection to visualSTATE model 281
 - events 283
 - events not sent to visualSTATE model. 285
 - external connections. 287
 - external input signals 280
 - external output signals 280
 - external signals 280–281, 283–284, 286
 - input signals. 284
 - interfacing with visualSTATE model 281
 - manipulating event generators not possible 285
 - opening existing designs 283
 - output object parameters 286
 - output objects. 284, 286
 - parameter values 287
 - power buttons. 286
 - Run mode. 285
 - saving connection bindings 285
 - simulating events 285
 - simulation 285
 - synchronization with visualSTATE model 289
 - toggle buttons. 286
 - unbinding objects and elements 285
 - unbound action functions. 283
 - unbound events 283
 - unbound inputs. 284
 - unused outputs 284
 - using parameters for external connections 286

- using with visualSTATE elements 279
- Altia Command Line Parameters option 289
- Altia connections 279–280, 286
 - closing 286
 - configuring 289
- Altia editor 283
- Altia FacePlate 279
- Altia menu 282
- Altia objects 284
- Altia parameters 286
- Altia Response Timeout 289
- ambiguous assignments 131, 133
- ambiguous behavior 119
- analysis
 - dynamic 198
 - static 183, 195
- Analyze command, Validator 365
- analyzing visualSTATE models 4, 149, 195, 365
- animation of objects, Altia 286
- API examples, OSEK 324
- API files
 - Basic, default configuration 243
 - Expert, default configuration 247
 - RealLink 209
- API functions 234
- API layers 234
- APIs 8–9, 164
 - code generation 233
 - stack sizes 330
- application programming interface 8
- arguments
 - for action function parameters 91
 - for custom commands 43
- arguments, for custom commands 43
- arithmetic 121
- arrowhead
 - double 164
 - red 162
- assigning events as conditions to breakpoints 171

- assigning expressions to breakpoints 172
- assigning signals as conditions to breakpoints 171
- assignments 165
 - activated 126
 - adding 94
 - dynamic ambiguous 131
 - static ambiguous 133
- assumptions in this guide xxvi
- automatic signal queue handling 168
- automatic simulation 149
- automatic vs. manual signal queue handling 167
- automatically generated code 9

B

- backup files 102
 - number of 102
- backup intervals 103
- Basic API
 - code generation 239
 - default configuration 243
- Basic API main function, example 211
- basic verification mode 114, 118
- basic verification mode, conflicting transitions 122
- baud rate 217
- binding external signals, Altia 280
- binding visualSTATE elements to Altia objects 284
- bindings, Altia 285
- bk files 102
- blank workspaces, creating 24
- bound external signals, Altia 283
- breaking execution of test sequences 192
- breakpoint conditions 169
 - searching for 192
- breakpoints 169, 174, 192
 - assigning conditions to 171
 - assigning expressions to 172
 - conditions 170
 - defining 170, 362

- deleting 180
 - disabling 170
 - enabling 170
 - evaluation of expressions 172
 - executed actions 174
 - expressions 172
 - for executed actions 174
 - graphical animation 180
 - pre-deduct conditions 174
 - setting for graphical animation 180
 - setting up for specific states 173
 - stepping over 174
 - stopping on 176
 - using 174
 - Breakpoints window 158
 - breaks 170, 175
 - breaks, macros performed on 224
 - building run-time applications, OSEK 321
 - button objects, Altia 280
- C**
- C header files, syntax 105
 - case-sensitivity, Coder command line 375
 - changing alias names 60
 - changing between Designer windows and views 343
 - changing explanation notes 60
 - changing state names 68
 - changing variable values 176
 - changing variable values, RealLink 223
 - checking in files, source code control 38
 - checking out files, source code control 38
 - checks by Verificator 123
 - activation of elements 125
 - conflicting transitions 128
 - dynamic ambiguous assignments 131
 - local dead ends 130
 - signal queue 133
 - state dead ends 129
 - static ambiguous assignments 133
 - System dead ends 131
 - unused elements 123
 - closing Altia connection 286
 - closing the Designer 106
 - closing the Navigator 35
 - closing Validator workspaces 152
 - code
 - readable. *See* human-readable code
 - sizes 249
 - table-based 239
 - user-written 8
 - code generation 237
 - Basic API 239
 - Expert API 245
 - starting 237
 - visualSTATE APIs 233
 - code generation information 235
 - code required for a visualSTATE application 9
 - Coder 4
 - Coder command line, case-sensitivity 375
 - Coder options 375
 - command line syntax 375
 - data width 249
 - lists of 375
 - rule data formats 250
 - setting 29
 - specifying keywords 250
 - specifying option files 375
 - Coder report files 235
 - Coder-generated code 8–9
 - Coder-generated files 240
 - header.h 240
 - SEMDef.h 240
 - SEMTypes.h 240
 - source.c 240
 - used in Basic default configuration 243
 - used in Expert default configuration 247
 - Coder-generated SEM type definitions 249

Coder-supported elements.	235	configurations	
COM.	217	Basic API, default	243
command line of Coder, case-sensitivity	375	Expert API, default	247
command line options, Verificator	371–372	configuring Altia connection.	289
command line parameters, Altia	289	configuring application, hints	81
command line syntax		configuring RealLink connection	215
Coder options.	375	configuring the Navigator	20
Documenter options	393	configuring the Validator for RealLink	215
Verificator	371	conflicting transitions	111–112, 121, 128
commands recorded to test sequence files.	188	basic verification mode	122
commands, user-specified	41	full verification mode.	122
communication devices, RealLink	213	guard verification mode	122
communication hardware, initialization of	212	connecting external signals, Altia	280–281
communication link, Altia	279	connecting visualSTATE elements to Altia objects	283–284
communication modules, RealLink	204	connection between Validator and target.	204
communication, RealLink.	206	connections, Altia	280
comparing outputs of test sequences.	193	connector states.	77
compatibility, rule data format numbers	251	connectors, Altia.	280, 284, 286
compilation time, hints for reducing	389	consistency, verification	113
completion transitions.	86	constants	104
complexity of verification		in existing files.	104
reducing.	144	unused	124
System configurations	145	verification.	123
use of operators	145	contradiction test code, disabling	388
use of signals and signal queues.	133, 144	controlling applications in target.	224
use of verification modes	144	controlling execution of code in target	225–226
composing states.	67	conventions in this guide.	xxvi
composing transitions.	83	conventions in this guide, verification.	109
composite states	72, 74	conversion of visualSTATE files	35
creating	72–73	conversion of visualSTATE files, manual	435
with concurrent regions	72	copies of Statechart files, creating.	100
compositional Systems	118	core model logic	240, 245
compositional verification mode.	114, 118	coverage (test)	
concurrent regions.	72–73	in percent	199
concurrent subsystems	75	creating blank workspaces	24
condition side	83	creating composite states	72–73
conditions		creating custom commands.	41
assigning to breakpoints.	171	creating elements	89
breakpoints	170, 192	creating graphical prototypes	279

- creating new Altia designs 283
 - creating new Projects in workspaces 25
 - creating parameters 90–91
 - creating Projects, Systems, and files in the Designer 97
 - creating software prototypes 291
 - creating state reactions 69
 - creating test sequences 190
 - creating Validator workspaces 152
 - creating workspaces 22
 - critical errors 135–136
 - current states 164
 - current states, retrieving 307
 - custom command arguments 43
 - custom command macros, renumbering 45
 - custom commands 41
 - activating 44
 - arguments for 43
 - creating 41
 - deleting 44
 - editing 44
 - Project-specific 42
 - prompt for arguments 44
 - renaming 44
 - silent mode 44
 - workspace-specific 42
 - customizing report layout 271
 - customizing the Designer 64
 - customizing the Navigator 20
 - C++ code, generating 237
 - C++ code, implementing visualSTATE code in 292
- D**
- data width 249
 - dead ends 129–131
 - debugging visualSTATE models 4, 149
 - declarations, action functions 246
 - declared elements 166
 - declaring action functions in external C files 93
 - deduction 174
 - deduction of events 305
 - deduction sequence 209
 - deep history states 78, 342
 - default configurations
 - Basic API 243
 - Expert API 247
 - default layout (statechart diagrams) 353
 - default names of RealLink functions, changing 213
 - default speed, test sequences 192
 - default values of options 31
 - defined events 153
 - defining Altia properties 289
 - defining breakpoint conditions 170
 - defining breakpoints 170, 362
 - defining elements 89
 - defining SEM type definitions 250
 - definitions configuring the Expert API 245
 - deleting breakpoints 180
 - deleting custom commands 44
 - deleting elements 89–90
 - deleting objects in statechart diagrams 60
 - deleting test sequences 190
 - design guidelines for verification 143
 - Designer 4
 - changing between windows and views 343
 - closing 106
 - customizing 64
 - importing files into 101
 - shortcuts 341
 - Designer backup files 102
 - Designer environment 49
 - Designer Project browser 50
 - Designer simulation mode 179
 - Designer toolbars 53–54
 - Designer windows 50
 - Designer-created files 37
 - detail level, generated visualSTATE Project reports 262
 - development with visualSTATE 7

device drivers	8–9
device drivers, generating	213
device drivers, MakeApp	104
diagram window	51
digital signature	33
digital signature, troubleshooting in RealLink	229
disabling breakpoints	170
disabling Systems	178
disconnecting objects and elements, Altia	285
display of warnings, setting	20
documentation report	4
Documenter	4
Documenter options	
command line syntax	393
online help	261
setting	29
Documenter-generated reports. <i>See</i> generated visualSTATE	
Project reports	
domains	121
double arrowhead	164
drawing states	58
drawing tools	349
drawing transitions	59
drop-if-full signal queue	134
dsn files	280
dynamic ambiguous assignments	131
dynamic analysis	197–198
dynamic analysis results	200
dynamic analysis, verification	113
dynamic formal verification	4
dynamic verification check	125
dynamically unread internal variables	126
dynamically unwritten internal variables	126

E

Edit mode, Altia	285
editing Altia designs	283
editing custom commands	44

editing elements	89
editing files under source code control	38
editing statechart diagrams	60
editor, Altia	283
editor, external	93–94
element browser	52
element searches, results of	95
element types, for transitions	84
elements	
checked	123
checked for activation	125
creating	89
declared	166
defining	89
deleting	89–90
editing	89
for transitions	85
not verified in full mode	117
not verified in guard mode	117
renaming	89
searching for	95
supported by RealLink	205
supported by the Coder	235
using with Altia	279
verified	123
embedded applications	8–9
emptying signal queues for specific Systems	169
emptying signal queues, manual	168
enabling breakpoints	170
enabling RealLink support	207
enabling Systems	178
environment, verification	114
error conditions, hints for tracking	81
errors	
because of signal queue overflow	134
critical	135
RealLink	228
verification	112
establishing connection to Altia design	281

- establishing RealLink connection 219
- evaluating breakpoint expressions 172
- event deduction 305
- event generators in Altia 285
- event groups
 - never activated 126
 - unused 124
- event inquiry 306
- event parameters
 - never activated 126
 - unused 124
 - verification 123
- event preprocessing 8
- event queues 8–9
- Event window 153
 - active events 164
 - filtering information 163
- events 284
 - activating 304
 - Altia 283
 - as conditions to breakpoints 171
 - defined 153
 - global 162
 - listing active 306
 - local 162
 - never activated 126
 - not sent from Altia to visualSTATE model 285
 - responding to 305
 - sending 167
 - sending inactive 162
 - simulating with Altia 285
 - unbound 283
 - unused 124
 - with parameters 162
- examples
 - mobile phone 302
 - traffic light system 287
 - visualSTATE Projects 4
 - VS OSEK API 324
- excluding states and regions from processing 80
 - overriding 81
- exclusion marks, overriding 81
- Expert API
 - code generation 245
 - default configuration 247
- Expert API files
 - default configuration 247
- Expert API main function, example 212
- Expert API requirements, RealLink 206
- Expert DLL 299
 - generating code for 301
 - implementing code in Visual Basic projects 302
 - interaction 300
 - interface files 300
 - interfacing to 302
 - mobile phone example 302
 - restrictions 300
- Expert DLL files 300
- ExpertR9.dll 300
- expressions, assigning to breakpoints 172
- extensions
 - dsn 280
 - oil 37
 - visualSTATE file names 407
 - vws 150
- external Altia connections 286–287
- external Altia signals 286
- external declarations, action functions 246
- external editor 93
- external input signals, Altia 280
- external logic 234
- external output signals, Altia 280
- external signals and Altia objects, connecting 281
- external signals, Altia 280–281, 283–284
- external variables
 - never activated 126
 - unused 123

F	
Field Chooser window, Validator	156
FIFO	167
file status, source code control	38
file types, source code control	37
files	
creating and saving in the Designer	97
from previous visualSTATE versions	35
importing into the Designer	101
input for generated visualSTATE Project reports	263
opening in the Designer	101
files from previous visualSTATE versions	435
files generated. <i>See</i> Coder-generated files	
files included in generated Project reports	258
filtering information in Event window	163
filtering information in System window	165
final states	342
finding elements	95
footers	346
forcing states	177
forcing Systems	177
fork states	78
formal test. <i>See</i> formal verification	
formal verification	110
Free Run	192
full verification mode	114–115
conflicting transitions	122
elements not verified	117
function declarations, in existing files	104
functionality test	149
G	
generated code, sizes	249
generated files. <i>See</i> Coder-generated files	
generated visualSTATE Project reports	257, 259
creating	258
customizing layout	271
included files	258
sections and detail level	262
setting options for	261
specifying contents	262
specifying files used as input	263
specifying HTML output format	267
specifying RTF output format	266
standard report layout	268
generating code	237
for Basic API	239
for Expert API	245
for Expert DLL	301
generating C++ code	237
generating device drivers	213
generating visualSTATE Project reports	258
getting started, Designer	57
getting started, visualSTATE	11
global events	162
graphical animation	179
Graphical Animation command	365
graphical animation options	180
graphical animation, setting breakpoints	180
graphical environment, Navigator	17
graphical prototypes, creating	279
graphical settings, Designer	65
graphical user interfaces, Visual Basic	299
Guard Expression window	155
guard expressions	163, 166
activated	125
adding	94
resolving	163
unresolved	166
values between deductions	166
guard verification mode	114–115
conflicting transitions	122
elements not verified	117
GUIs, in Visual Basic	299

H

handling of signal queues, manual 168
 handling visualSTATE files from previous versions 435
 Harvard architecture, RealLink 206, 208
 headers 346
 header.h 240
 hierarchical state machines 75
 history states. *See* shallow history states
 HTML output format, generated Project reports 267
 HTML page shown at start up, changing 20
 human-readable code 241
 size 253
 human/machine interface 277

I

implementation of functions, RealLink examples 213
 implementation, prototypes 292
 implementing action functions in external C files 93
 implementing code in Visual Basic projects 302
 implementing target-specific functions, RealLink 213
 implementing visualSTATE code in C++ code 292
 importing files into the Designer 101
 importing Projects into workspaces 27
 inactive events, sending 162
 Include Guard Expressions command 164
 information about code generation 235
 information in Event window, filtering 163
 information in System window, filtering 165
 initial states 78
 initialization, Systems 364
 initializing priority queues 303
 initializing Systems 162
 input signals, Altia 280, 284
 inputs 280
 instances, activating 178
 instances, setting up order of 178
 instances, specifying number of 101

integrating visualSTATE code with user-written code 9
 interaction, visualSTATE Expert DLL 300
 interactive simulation 161, 178, 188
 interface, verification 114
 interfacing to the Expert DLL using Visual Basic 302
 interfacing visualSTATE model with Altia design 281
 internal logic 234
 internal variables
 dynamically unread 126
 dynamically unwritten 126
 statically unread 123
 statically unwritten 123
 interrupt functions, generating 213
 intervals, backup 103

J

join states 78
 junction states 79

K

keywords, specifying 250

L

large state spaces, verification 113
 layout, statechart diagrams 353
 LED objects 280
 level of detail, verification 114
 limitations when using Expert DLL 300
 listing active events 306
 livelock 168
 loading visualSTATE Systems 303
 local dead ends 112, 130
 local events 162
 location of visualSTATE user documentation files 20
 logic
 external 234

internal	234
visualSTATE Systems	51, 240, 246
logical consistency	113
login, source code control systems	39
login, Windows	39

M

macrosteps	224
macros, VS_WAIT()	224
macros, where breaks are performed	224
main function	210–212
main loop	210
MakeApp device drivers	104
manipulating event generators not possible, Altia	285
manual conversion of files from previous VS versions	435
manual emptying of signal queues	168
manual signal queue handling	168
manual simulation	188
manual vs. automatic signal queue handling	167
mapping MakeApp device drivers	104
margins, statechart print-outs	346
memory consumption, RealLink	409–410
memory usage, RealLink	409–410
memory, RealLink	206
menus, Altia	282
Microsoft Common Source Code Control	37
Microsoft SCC API	37
microsteps	224
mobile phone example	302
model dependent memory usage, RealLink	409
modeling guidelines for verification	143
models	279
analyzing	195
verification	114
modes of verification	114
basic mode	114, 118
compositional mode	114, 118
differences	114

full mode	114–115
guard mode	114–115
monitoring target applications	220
monitoring visualSTATE elements	222
Move cursor	63
MultiUser Management	37
mutually exclusive substates	74

N

navigating in statechart diagrams	60
Navigator	3
closing	35
customizing	20
reloading files	32
Navigator graphical environment	17
Navigator output window	18
Navigator properties window	19
Navigator toolbars	19
Navigator windows	18
Navigator workspace browser	18
Navigator workspaces	21
Navigator workspaces, saving	338
Navigator-created files	37
never activated elements	111
never activated event groups	126
never activated event parameters	126
never activated events	126
never activated external variables	126
never activated transitions	126
never sent signals	124, 126
non-verifiable elements	119
notes, in statecharts	60
number of backup files	102
N/A	166

O

objects in statechart diagrams	62
--	----

- objects, Altia 284
 - off-page regions 75
 - OIL 314
 - oil files 37
 - online help 5, 31
 - online help, Documenter options 261
 - opening existing Altia designs 283
 - opening Projects in the Designer 101
 - opening test sequences 191
 - opening Validator workspaces 152
 - opening workspaces 25
 - operating systems 313
 - operators 145
 - optimization of SEM type definition sizes 249
 - optimization, generated code 249
 - option files 375
 - options
 - default values 31
 - for Coder 375
 - for generated Project reports 261
 - graphical, in the Designer 65
 - order
 - of signal queue emptying 168
 - of Systems/instances 178
 - OSEK 313, 321
 - building run-time applications 321
 - OIL 314
 - running visualSTATE OSEK wizard 315
 - stack usage 329
 - supplying events 323
 - visualSTATE API examples 324
 - visualSTATE OSEK API functions 323
 - OSEK environment, using visualSTATE files in 313
 - OSEK support, enabling 313
 - OSEK tasks 315
 - output object parameters, Altia 286
 - output objects, Altia 284, 286
 - output signals, external 280
 - output types 189
 - output window, Navigator 18
 - output window, Validator 157
 - outputs 280
 - outputs of steps recorded to a test sequence file 188
 - outputs of test sequences, comparing 193
 - outputs produced by sent events 165
- ## P
- pairs of states 77
 - parameter values, visualSTATE elements and Altia 287
 - parameters 162
 - Altia 286
 - Altia output objects 286
 - creating 90–91
 - parity 217
 - pausing execution of recorded test sequences 192
 - playing recorded test sequences 191
 - playing sequences of target tests 227
 - positioning objects in statechart diagrams 63
 - power buttons, Altia 286
 - preparing target application for using RealLink 207
 - previous visualSTATE versions, files from 35
 - printing statechart diagrams 63
 - priority queues, initializing 303
 - processors, RealLink 206
 - Project browser, Designer 50
 - Project examples 4
 - Project view 51
 - Projects 300
 - creating and saving in the Designer 97
 - creating new in workspaces 25
 - importing into a workspace 27
 - in workspaces 22
 - opening in the Designer 101
 - removing from workspaces 28
 - restrictions when using Expert DLL
 - setting as active 28
 - setting up 12

Project-specific custom commands	42
Project-specific files	240, 245
prompt for arguments, custom commands	44
properties window, Navigator	19
properties, RealLink	216
prototype implementation	292
prototypes	277
prototypes, creating graphical	279
prototyping	277
prototyping, with visualSTATE Expert DLL	299
pseudostates	77

Q

queues of signals	225
-------------------	-----

R

RAM/ROM usage, visualSTATE OSEK API	333
reachable transitions	125
readable code. <i>See</i> human-readable code	
RealLink	4, 180, 222, 226
accessing target communication device	213
changing variable values	223
communication	206
communication modules	204
configuration of Validator	215
controlling applications in target	224
controlling execution of code in target	225–226
errors	228
Harvard architecture	206, 208
implementation of functions, examples	213
implementation of Receive function	215
implementation of Transmit function	214
implementing target-specific functions	213
initialization of communication hardware	212
location of visualSTATE Systems	213
manipulating target application	223
memory	206

memory consumption, with Basic API	410
model dependent memory usage	409
monitoring target applications	220
monitoring visualSTATE elements	222
preparing target application	207
properties	216
receive functions	206
RS232 setup	217
sending events into target	224
setting up	207
setting up RS232 communication	217
setting up TCP/IP communication	218–219
setup	204
supported visualSTATE elements	205
target processors	206
target requirements	206
target-specific functions	213
troubleshooting	228–229
Validator windows	220
variable sizes	206
visualSTATE Expert API requirements	206
VS_WAIT()	220
RealLink API dependent memory usage	410
RealLink API files	209
RealLink API, using	210
RealLink components	204
RealLink connection	204
configuring	215
establishing	219
setting up	215
RealLink functions	213, 215
changing default names of	213
Receive()	213
Reset()	213
TransmitFlush()	213
Transmit()	213
RealLink memory consumption	409
RealLink support file	210
RealLink support, enabling	207

RealLink.c file 209
 RealLink.h file 209
 real-time operating systems 235, 313
 receive functions, RealLink 206
 Receive(), RealLink function 213, 215
 recorded test sequences 192
 recorded test sequences, playing 191
 recording sequences of target tests 227
 recording test sequences 187
 recording to test sequence files 365
 red arrowhead 162
 reducing complexity of verifying Systems 144
 regions 75
 concurrent 72–73
 contents of 75
 excluding from processing 80
 overriding 81
 in topstates 76
 off-page 75
 reloading files in the Navigator 32
 removing Projects from workspaces 28
 renaming custom commands 44
 renaming elements 89
 renaming objects in statechart diagrams 60
 renumbering of custom command macros 45
 report contents, specifying 262
 report files, Coder 235
 report layout
 customizing 271
 standard 268
 reports on visualSTATE Projects. *See* generated visualSTATE
 Project reports
 reset event 318
 reset event name 162
 Reset(), RealLink function 213
 resizing objects in statechart diagrams 63
 resolving guard expressions 163
 responding to events 305
 restoring options to default values 31
 restrictions when using Expert DLL 300

results
 of dynamic analysis 200
 of element searches 95
 retrieving copies of files under source code control 38
 retrieving current states 307
 return values of actions 177
 RL_TCPIP.cpp file 218
 route points 59
 RS232 communication for RealLink, setting up 217
 RS232 communication plugin, troubleshooting 229
 RS232 setup, RealLink 217
 RTF output format, generated Project reports 266
 RTOS 235
 rule data format numbers
 visualSTATE Classic version 3 compatibility 251
 visualSTATE Pro version 3 compatibility 251
 rule data formats 250
 rule data header word types 250
 rule data header word width 250
 rule data width 250
 Run mode, Altia 285
 run-time applications, building for OSEK 321
 run-time models, viewing 180

S

safe mode 64, 119
 sample code, visualSTATE 5
 saving connection bindings, Altia 285
 saving Navigator workspaces 338
 saving Projects, Systems, and files in the Designer 97
 saving workspaces 22, 338
 searching for breakpoint conditions 192
 searching for elements 95
 sections, of generated visualSTATE Project reports 262
 selecting objects in statechart diagrams 62
 SEM type definitions
 Coder-generated 249
 defining 250

forcing width of	249	manual emptying	168
sizes	249	manual vs. automatic handling	167
SEMBDef.h	240	not empty	167
SEMTypes.h	240	single-stepping	168
sending events into target, RealLink	224	stepping	169
sending inactive events	162	unbounded	135
Set Next Step command	192	verification	133, 144
setting action function return values	177	signals	165
setting breakpoints for graphical animation	180	Altia	280
setting Coder options	29	as conditions to breakpoints	171
setting Documenter options	29	external	280
setting Projects as active	28	never sent	124, 126
setting speed of test sequence execution	192	never used as triggers	124, 126
setting Systems as active	28	stop sending of	168
setting up breakpoint conditions	170	verification	133, 144
setting up breakpoints for executed actions	174	silent mode, custom commands	44
setting up custom commands	41	simple states, drawing	58
setting up order of Systems/instances	178	simulating visualSTATE action functions	285
setting up RealLink	207	simulation	4, 149, 161, 164
setting up RealLink connection	215	Altia	285
setting up visualSTATE Projects	12	automatic	149
setting Verificator options	29	interactive	161, 178
settings, graphical in the Designer	65	manual	188
setup, RealLink	204	simulation modes, Validator	149
setup, Systems	178, 363	simulation mode, Designer	179
SE_RESET	162, 318	single-stepping signal queues	168
shallow history states	78, 342	sizes	
shortcut keys, Validator	355	of generated code	249
shortcuts, Designer	341	of SEM type definitions	249
signal queue behavior, specifying	95	optimization, SEM type definitions	249
signal queue handling	156	software prototypes	291
automatic	168	source code control	
manual	168	adding files	37
signal queue overflow	133	checking in files	38
signal queue size, specifying	95	checking out files	38
Signal Queue window	155–156, 165	editing files	38
signal queues	156, 165, 225	file status	38
automatic handling	168	retrieving copies of files	38
emptying for specific Systems	169	supported visualSTATE file types	37

- using 37
- source code control systems 37
 - accessing files 38
 - user name 39
- source code editor, external 94
- source.c 240
- Specify bits for encoding variables option 121
- specifying keywords 250
- specifying number of System instances 101
- specifying option files 375
- specifying report contents 262
- speed of test sequence execution, setting 192
- speeding up verification of compositional Systems 118
- speed, test sequences 192
- Split readable code option 389
- stack sizes, visualSTATE APIs 330
- stack usage 329
- standard report layout 268
- starting code generation 237
- starting verification 137, 141
- starting visualSTATE programs 11
- startup state, initializing System to 364
- state dead ends 112, 129
- state machines, hierarchical 75
- state names, changing 68
- state reactions, creating 69
- state spaces, verification 113
- statechart diagrams 57, 62
 - aligning and resizing objects in 63
 - default layout 353
 - deleting objects in 60
 - editing 60
 - navigating 60
 - printing 63
 - selecting objects in 62
- Statechart files 21
- Statechart files, creating copies of 100
- statechart notes 60
- states
 - composing 67
 - drawing 58
 - excluding from processing 80
 - overriding 81
 - forcing 177
 - never activated 125
 - pairs of 77
 - retrieving current 307
 - setting up breakpoints for 173
 - unused 123
- static ambiguous assignments 133
- static analysis 183, 195
- statically unread internal variables 123
- statically unwritten internal variables 123
- status of files, source code control 38
- stepping signal queues 169
- stepping test sequences 364
- steps, in test sequences 187
- stop points 192
- stop sending signals 168
- stopping recording to test sequence files 365
- stopping timers 158
- substates 74
- subsystems, concurrent 75
- summary information 235
- supported visualSTATE file types, source code control 37
- synchronization, with Altia design 289
- syntax, C header files 105
- syntax, option files 375
- System configurations, and complexity of verification 145
- System dead ends 112, 131
- System instances, specifying number of 101
- System setup 178
- System Setup window 156
- System tools 350
- System view, Designer 51
- System window 152
- System window, filtering information in 165

Systems	21
commands recorded in test sequence files	188
compositional	118
core model logic	245
creating and saving in the Designer	97
disabling	178
enabling	178
forcing	177
initialization	364
initializing	162
loading	303
location when using RealLink	213
logic	51, 240, 246
order	178
setting as active	28
setting up order of	178
setup	363
unloading	308
System-specific files	240

T

table-based code	239
target applications, manipulating	223
target mode	150, 180
target processors, RealLink	206
target requirements, RealLink	206
target tests, recording and playing sequences of	227
target-specific RealLink functions	213
TCP/IP communication for RealLink, setting up	218–219
test coverage	197
in percent	199
test sequence execution, speed	192
test sequence files	188
deleting current sequences	191
outputs of steps	188
recording	365
stopping recording	365

test sequences	187, 190, 192–193
breaking execution	192
creating	190
default speed	192
deleting	190
opening	191
playing	191
recording	187
stepping	364
Timers window	158
timers, stopping	158
toggle buttons	286
toolbars	
Designer	53–54
Navigator	19
Validator	158
topstates, with regions	76
tracing visualSTATE models	149, 183
traffic light system, example	287
transition description	59
transition elements	84–85
transitions	
composing	83
conflicting	128
drawing	59
never activated	126
reachable	125
without triggers	86
TransmitFlush(), RealLink function	213
Transmit(), RealLink function	213–214
troubleshooting	
event generators in Altia cannot be manipulated	285
events not sent from Altia	285
RealLink	228
type definitions	240, 245
types of outputs	189

U

unbinding objects and elements, Altia	285
unbound action functions, Altia	283
unbound events, Altia	283
unbounded signal queues	135
unloading visualSTATE Systems	308
unresolved variables	166
unused action functions	124
unused Altia inputs	284
unused Altia outputs	284
unused constants	124
unused elements	123
unused event groups	124
unused event parameters	124
unused events	124
unused external variables	123
user documentation files, setting location	20
user documentation, visualSTATE	5
user name, source code control systems	39
user name, Windows	39
user-specified commands	41
user-written code	8–9
using breakpoints	174
using the RealLink API	210

V

Validator	4
Validator Altia connection, configuring	289
Validator Analyze command	365
Validator environment	150
Validator mode	149, 180
Validator output window	157
Validator shortcut keys	355
Validator toolbars	158
Validator tools	149
Validator Watch window	157, 222
Validator windows	152
Validator windows and RealLink	220
Validator workspace files	151
Validator workspaces	150
Validator workspaces, closing	152
Validator workspaces, creating	152
Validator workspaces, opening	152
Validator-created files	37
variable sizes, RealLink	206
variable values, changing	176
Variable window	155
variables	
unresolved	166
verification	121, 123
verification	110
activated action functions	126
activated assignments	126
activated guard expressions	125
ambiguous behavior	119
arithmetic	121
basic mode	114, 118
compared with simulation	110
compositional mode	114, 118
conflicting transitions	121
critical errors	135–136
design guidelines	143
domains	121
dynamic	4
dynamic analysis	113
dynamically unread internal variables	126
dynamically unwritten internal variables	126
elements checked for activation	125
elements not verified in full mode	115, 117
elements not verified in guard mode	115, 117
environment	114
errors	112
full mode	115
guard mode	114–115
interface	114
large state spaces	113

level of detail	114	unused elements.	123
logical consistency.	113	verification errors	135
model.	114	verification modes.	114, 135
never activated event groups	126	basic mode.	114, 118
never activated event parameters	126	complexity of verification	144
never activated events	126	compositional mode.	114, 118
never activated external variables.	126	conflicting transitions	122
never activated states	125	differences	114
never activated transitions	126	full mode	114–115
non-verifiable elements	119	guard mode	114–115
of compositional Systems	118	Verificator	4
reducing complexity of verifying Systems	144	Verificator command line options	371–372
signal queue overflow	133	Verificator command line syntax.	371
signals	133	Verificator options, setting	29
signals never sent.	124, 126	Verificator, activating	137, 141
signals never used as triggers.	124, 126	verified elements.	123
starting.	137, 141	viewing copies of files under source code control.	38
statically unread internal variables.	123	viewing generated visualSTATE Project reports	259
unused action functions	124	viewing run-time models	180
unused constants	124	Visual Basic	299
unused event groups.	124	Visual Basic projects, implementing code.	302
unused event parameters	124	Visual Basic, interfacing to Expert DLL.	302
unused events.	124	visualSTATE APIs. <i>See</i> APIs	
unused external variables.	123	visualSTATE application, code required	9
unused states	123	visualSTATE Coder. <i>See</i> Coder	
variables.	121	visualSTATE code, implementing in C++ code	292
variables read.	123	visualSTATE deduction sequence	209
variables written.	123	visualSTATE Designer. <i>See</i> Designer	
visualSTATE generated code	146	visualSTATE development	7
warnings and errors	112	visualSTATE Documenter. <i>See</i> Documenter	
verification checks	123, 135	visualSTATE elements supported by RealLink	205
activation of elements	125	visualSTATE embedded applications	8–9
conflicting transitions.	128	visualSTATE Expert DLL. <i>See</i> Expert DLL	
dynamic ambiguous assignments.	131	visualSTATE file name extensions	407
local dead ends.	130	visualSTATE files from previous versions	35, 435
signal queues	133	visualSTATE files, for use in OSEK environment.	313
state dead ends	129	visualSTATE global layers	234
static ambiguous assignments	133	visualSTATE layers.	234
System dead ends.	131	visualSTATE local layers	234, 240, 245

visualSTATE MakeApp files 104
 visualSTATE models, analyzing 195
 visualSTATE models, interfacing with Altia designs 281
 visualSTATE modules 3
 visualSTATE MultiUser Management 37
 visualSTATE Navigator. *See* Navigator
 visualSTATE OSEK API 329
 visualSTATE OSEK API functions 323
 visualSTATE OSEK API, RAM/ROM usage 333
 visualSTATE OSEK Kit 313
 building run-time applications 321
 visualSTATE OSEK wizard 315
 visualSTATE programs 3
 visualSTATE Project examples 4–5
 visualSTATE Project reports. *See* generated visualSTATE
 Project reports
 visualSTATE Projects. *See* Projects
 visualSTATE RealLink. *See* RealLink
 visualSTATE reset event 318
 visualSTATE sample code 5
 visualSTATE software 3
 visualSTATE Statechart files. *See* Statechart files.
 visualSTATE Systems. *See* Systems
 visualSTATE user documentation 5
 visualSTATE Validator. *See* Validator
 visualSTATE Verificator. *See* Verificator
 visualSTATE, previous versions 435
 visualSTATE, starting 11
 VS Project-specific files 240, 245
 VS System-specific files 240
 VSrtps.c file 210
 VS_WAIT() 210, 220, 224
 vws files 150

W

warnings 365
 warnings, verification 112
 Watch window, Validator 157, 222

width of SEM type definitions, forcing 249
 Windows user name 39
 windows, Navigator 18
 windows, Validator 152
 wizard, for creating a Navigator workspace 22
 workspace browser, Navigator 18
 workspace files, Validator 151
 workspace wizard, Navigator 22
 workspaces 21
 creating 22, 24
 creating new Projects in 25
 importing Projects into 27
 opening 25
 Projects in 22
 removing Projects from 28
 saving 22, 338
 saving in Navigator 338
 workspaces, Validator 150
 workspace-specific custom commands 42
 wrap-around 121

Z

zoom view 53
 zooming 343