IAR
Visual State

IAR Visual State

User Guide

UVS-13

IAR SYSTEMS

EDITION NOTICE

Thirteenth edition: January 2019

Part number: UVS-13

This guide applies to version 10.1.x of IAR Visual State.

This guide replaces the obsolete guides *IAR visualSTATE® User Guide* (UVS-5 and previous editions), as well as the *IAR visualSTATE® API Guide*, the *IAR visualSTATE® Version 5 Project Setup Guide*, *IAR visualSTATE® Reference Guide*, *IAR visualSTATE® Quick Start Tutorial*, *C-SPY®Link User Guide*, and *Getting Started with visualSTATE®*.

Internal reference: IJOA.

Brief contents

Contents

Tables

Preface

Welcome to the *IAR Visual State User Guide*. This guide describes how to use IAR Visual State to develop and test embedded applications based on state machines.

For information about installation, see the *Installation and Licensing Quick Reference* booklet—available in the product box—and the *Licensing Guide*.

Who should read this guide

Read this guide if you plan to develop an application based on state machines using IAR Visual State.

REQUIRED KNOWLEDGE

To use the tools in IAR Visual State, you should have working knowledge of:

- The architecture and instruction set of the microprocessor core that you are using (refer to the chip manufacturer's documentation)
- The programming language of the generated source code. Visual State can generate C, C++, C#, and Java source code.
- Application development for embedded systems
- Basic principles of state/event modeling
- The operating system of your host computer.

How to use this guide

Each part in this guide covers a specific *topic*. In each part, the information is typically divided into chapters based on *information types*:

- *Concepts*, which describes the topic and gives overviews of features related to the topic. Any requirements or restrictions are also listed. Read this information to learn about the topic.
- *Tasks*, which lists useful tasks related to the topic. For many of the tasks, you can also find step-by-step descriptions. Read this for information about required tasks as well as for information about how to perform certain tasks.

- *Reference information*, which gives reference information related to the topic. Read this section for information about certain GUI components. You can easily access this type of information for a certain component in the GUI by pressing F1.

The tutorials in the IAR Information Center will help you get started using IAR Visual State.

Finally, we recommend the *Glossary* if you should encounter any unfamiliar terms in the IAR Systems user documentation.

What this guide contains

Below is a brief outline and summary of this guide.

PART 1. IAR VISUAL STATE AND ITS COMPONENTS

- *IAR Visual State and state machine design* gives an introduction to IAR Visual State and its components, and why you should use state machines to design your embedded application.

PART 2. PROJECT MANAGEMENT USING THE NAVIGATOR

- *Project management* gives an introduction to project management using the Visual State Navigator, as well as information about related tasks, including step-by-step descriptions. The chapter also contains reference information for the related graphical environment.
- *The IAR Visual State Compare Tool* describes how to visualize differences between two versions of a state machine model or complete project.
- *Custom commands* gives an introduction to using custom commands, as well as information about related tasks, including step-by-step descriptions. The chapter also contains reference information for the related graphical environment.

PART 3. DESIGNING USING THE DESIGNER

- *Designing* gives an introduction to designing state machines using the Visual State Designer, as well as information about related tasks, including step-by-step descriptions.
- *States* gives an introduction to states, as well as information about related tasks, including step-by-step descriptions.
- *Transitions* gives an introduction to transitions, conditions, and actions, as well as information about related tasks and to some of them also step-by-step descriptions.
- *Transition elements* gives an introduction to transition elements, such as events, event groups, signals, and action functions, as well as information about related

tasks, including step-by-step descriptions. The chapter also contains reference information about Visual State operators and operands.

- *Reusing designs using state machine templates* gives an introduction to how to reuse designs using state machine templates and submachine states, as well as information about related tasks, including step-by-step descriptions.
- *Using variants and features* gives an introduction to how to design your product as multiple similar variants, for example as a Premium version and a Basic version or as versions for different sales regions, to avoid having to maintain two or more separate software development tracks.
- *Using requirements files* explains how to import formal design requirements in a standardized format called ReqIF (Requirements Interchange Format) and how to tie objects in your Visual State designs to corresponding requirements, to keep track of how your design fulfills all or some of the requirements.
- *The Visual State Designer* gives an introduction to the Visual State Designer, as well as information about related tasks, including step-by-step descriptions. The chapter also contains reference information about the related graphical environment, as well as for the syntax for C header files.

PART 4. SIMULATING USING THE VALIDATOR

- *Simulation* gives an introduction to simulating state machine models using the Visual State Validator, as well as information about related tasks, including step-by-step descriptions.
- *Graphical animation* gives an introduction to graphically animated debug sessions, as well as information about related tasks, including step-by-step descriptions. The chapter also gives reference information for the related graphical environment.
- *Tracing* gives an introduction to tracing in state machines, as well as information about related tasks, including step-by-step descriptions.
- *Analyzing* gives an introduction to analyzing your design model by performing either static or dynamic analysis, as well as information about related tasks, including step-by-step descriptions.
- *Recording and playing test/event sequences* gives an introduction to recording and playing your test sequences, as well as information about related tasks and to some of them also step-by-step descriptions. The chapter also gives a syntax description of the event sequence file.
- *The Visual State Validator* gives an introduction to the Visual State Validator. The chapter also contains reference information about the related graphical environment.

PART 5. FORMAL VERIFICATION USING THE VERIFICATOR

- *Formal verification* gives an introduction to formal verification using the Visual State Verificator, as well as information about related tasks, including step-by-step descriptions. The chapter also contains reference information about the related graphical environment.
- *Checks performed by the Verificator* gives an overview of available checks, modes, and errors, as well detailed information about the checks and how to perform them.
- *Verificator command line options* describes how to invoke the Verificator using command line options. The chapter also contains reference information about the Verificator command line options.

PART 6. CODE GENERATION USING A CODER

- *Code generation* gives an introduction to code generation and the Visual State APIs, as well as information about related tasks, including step-by-step descriptions.
- *HCoder API code generation* gives an introduction to the HCoder API code generation, as well as information about related tasks, including step-by-step descriptions.
- *HCoder API reference information* gives an overview of the coder-generated source files for the HCoder API, as well as reference information about the HCoder API functions and return codes.
- *The Visual State Hierarchical Coder* gives an introduction to the Visual State Hierarchical Coder. The chapter also contains reference information about the related graphical environment, the type identifiers, and the transition rule data format.
- *Hierarchical Coder command line options* describes how to invoke the Hierarchical Coder using command line options. The chapter also contains reference information about the Hierarchical Coder command line options.
- *Adaptive API code generation* gives an introduction to the Adaptive API code generation, as well as information about related tasks, including step-by-step descriptions.
- *Uniform API code generation* gives an introduction to the Uniform API code generation, as well as information about related tasks, including step-by-step descriptions.
- *Adaptive API reference information* gives an overview of the coder-generated source files for the Adaptive API, as well as reference information about the Adaptive API functions and return codes.
- *Uniform API reference information* gives an overview of the coder-generated source files for the Uniform API, as well as reference information about the Uniform API functions and return codes.

- *The Visual State Classic Coder* gives an introduction to the Visual State Classic Coder. The chapter also contains reference information about the related graphical environment, the SEM type identifiers, and the transition rule data format.
- *Classic Coder command line options* describes how to invoke the Classic Coder using command line options. The chapter also contains reference information about the Classic Coder command line options.

PART 7. TESTING YOUR STATE MACHINE MODEL ON HARDWARE

- *Debugging design models using C-SPYLink* gives an introduction to C-SPYLink, as well as information about related tasks, including step-by-step descriptions. The chapter also contains reference information for the related graphical environment.
- *Debugging design models using RealLink* gives an introduction to RealLink, as well as information about related tasks, including step-by-step descriptions. The chapter also contains reference information for the related graphical environment.

PART 8. DOCUMENTING VISUAL STATE PROJECTS USING THE DOCUMENTER

- *Documenting projects* gives an introduction to documenting projects using the Documenter, as well as information about related tasks, including step-by-step descriptions. The chapter also contains reference information for the related graphical environment.
- *Documenter command line options* describes how to invoke the Documenter using command line options. The chapter also contains reference information about the Documenter command line options.

PART 9. ADDITIONAL FEATURES AND UTILITIES

- *Prototyping a graphical interface* gives an introduction to prototyping a graphical interface, either by using the built-in support in the Validator for connecting to Altia Design, or by integrating Visual State Coder-generated code with code developed in a third-party development tools to create a graphical model.
- *Viewing design models via the Visual State Viewer* gives an introduction to the Visual State Viewer and how to use it for viewing state machine models.
- *Using IAR Visual State remotely via the Control Center* gives an introduction to using visual state remotely via the Control Center, as well as information about related tasks, including step-by-step descriptions.
- *Importing and exporting design models via XMI® files* gives an introduction to the XMI file format and how to use it for importing and exporting state machine models between IAR Visual State and tools from other vendors. The chapter also provides information about related tasks, including step-by-step descriptions.

- *The Visual State State Machine API for programmatic manipulation of models* gives an introduction to programmatically manipulating models using the Visual State State Machine API.
- *Handling Visual State files from previous versions* describes tasks related to converting Visual State files from previous versions.
- *Glossary* lists terms relevant to embedded systems programming in general, and to IAR Visual State and state machine design in particular.

Other documentation

User documentation is available as hypertext PDFs and as a context-sensitive online help system in HTML format. You can access the documentation from the Information Center or from the **Help** menu in IAR Visual State. The online help system is also available via the F1 key.

USER AND REFERENCE GUIDES

The complete set of IAR Systems development tools is described in a series of guides. Information about:

- System requirements and information about how to install and register the IAR Systems products, is available in the *Installation and Licensing Quick Reference* booklet—available in the product box—and the *Licensing Guide*.
- Using IAR Visual State for developing and testing embedded applications based on state machines, is available in the *IAR Visual State User Guide* (this guide).

THE ONLINE HELP SYSTEM

The context-sensitive online help contains:

- Information about concepts related to using IAR Visual State and its components
- Information about how to perform certain tasks
- Reference information about the graphical environment, such as menus, windows, and dialog boxes
- Reference information about the API functions and command line options

WEB SITES

Recommended web sites:

- The chip manufacturer's web site that contains information and news about the microcontroller core you are using.

- The IAR Systems web site, **www.iar.com**, that holds application notes and other product information.
- The Object Management Group® consortium web site for the UML standard, **www.uml.org**.

Document conventions

When, in the IAR Systems documentation, we refer to the programming language C, the text also applies to C++, unless otherwise stated.

When referring to a directory in your product installation, for example `\doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\Visual State 8.`*`n`*`\doc`.

TYPOGRAPHIC CONVENTIONS

The IAR Systems documentation set uses the following typographic conventions:

Style	Used for
`computer`	• Source code examples and file paths. • Text on the command line. • Binary, hexadecimal, and octal numbers.
`parameter`	A placeholder for an actual value used as a parameter, for example *`filename`*`.h` where *`filename`* represents the name of the file.
`[option]`	An optional part of a command.
`[a|b|c]`	An optional part of a command with alternatives.
`{a|b|c}`	A mandatory part of a command with alternatives.
bold	Names of menus, menu commands, buttons, and dialog boxes that appear on the screen.
italic	• A cross-reference within this guide or to another guide. • Emphasis.
`...`	An ellipsis indicates that the previous item can be repeated an arbitrary number of times.
	Identifies instructions specific to the Visual State Navigator interface.
	Identifies instructions specific to the command line interface.
	Identifies helpful tips and programming hints.

Table 1: Typographic conventions used in this guide

Style	Used for
	Identifies warnings.

Table 1: Typographic conventions used in this guide (Continued)

NAMING CONVENTIONS

The following naming conventions are used for the products and tools from IAR Systems®, when referred to in the documentation:

Short form	Refers to
Navigator	Visual State Navigator
Designer	Visual State Designer
Validator	Visual State Validator
Verificator	Visual State Verificator
Coder	Visual State Coder
Documenter	Visual State Documenter
Adaptive API	Visual State Adaptive API
Uniform API	Visual State Uniform API
project	Visual State project
system	Visual State system

Table 2: Naming conventions used in this guide

Part 1. IAR Visual State and its components

This part of the *IAR Visual State User Guide* includes these chapters:

- IAR Visual State and state machine design

IAR Visual State and state machine design

- Introduction to IAR Visual State and its components
- Application development using IAR Visual State

Introduction to IAR Visual State and its components

Learn more about:

- *Why use IAR Visual State and state machines*, page 55
- *IAR Visual State overview*, page 56
- *Important features and advantages*, page 59

WHY USE IAR VISUAL STATE AND STATE MACHINES

State machines are commonly used for describing discrete systems, where the current behavior is a result of previously occurring events.

A *state machine* consists of a hierarchy of states and transitions between the states, which you create by drawing *state machine diagrams*. Because a diagram is a graphical representation which is easy to create, understand, communicate, and change, there are several important design advantages to organizing the logic of your application this way.

State machines allow you to develop the specification and application in a natural, iterative way where states in the machine corresponds to states in your application. You get a high-level view that helps you handle the complexity of the application. You can outline your application and then add functionality at a more detailed level, step by step.

One very important feature of a state machine is its ability to handle concurrency. You can model concurrent behavior without necessarily having to involve more than one task (or process if an operating system is used). It might even eliminate the need for an operating system in some situations.

Once the state machine diagram has been created, its logic can be tested and verified to make sure the state machine behaves as intended.

State machines are very useful for controlling logic-oriented applications where reliability, size, and deterministic execution are the main requirements.

As an example of a state machine, consider a vending machine and all the cases that must be considered:

- What happens if a cup is removed before it is full?
- What happens if a new order is started before the previous order has been completed?
- Will the money be correctly returned to the customer if one of the electromechanical parts causes the machine to stop in the middle of processing an order?

For more information about state machines and state machine diagrams, see *Introduction to designing state machines using the Designer*, page 117.

IAR VISUAL STATE OVERVIEW

IAR Visual State is a set of fully integrated development tools for designing, testing, and implementing embedded applications based on state machine models. It includes a graphical design environment, verification and validation tools for testing, a code generator, and a documenter for documenting your design:

Component	Description
Navigator	The Visual State Navigator is a graphical project management tool for your Visual State projects. From the Navigator you access and activate the other components in IAR Visual State, and set options for the Verificator, Coder and Documenter. See *Part 2. Project management using the Navigator*, page 69.
Designer	The Visual State Designer is a graphical tool for designing state machines by drawing state machine diagrams using the UML notation. See *Part 3. Designing using the Designer*, page 115.
Validator	The Visual State Validator is a graphical tool for simulating, analyzing, and debugging models created with the Designer. Use the Validator to test the functionality of your design. See *Part 4. Simulating using the Validator*, page 319.
Verificator	The Visual State Verificator is a tool for dynamic formal verification of models created with the Designer. See *Part 5. Formal verification using the Verificator*, page 411.
Coders	The Visual State Coders automatically generate code for the models created with the Designer, which is to be combined with your manually written code. See *Part 6. Code generation using a Coder*, page 455.

Table 3: IAR Visual State components overview

Component	Description
Documenter	The Visual State Documenter creates up-to-date documentation reports for your Visual State project, including design, tests, and code generation. See *Part 8. Documenting Visual State projects using the Documenter*, page 811.
Viewer	The Visual State Viewer is a stand-alone application for viewing Visual State state machine models, without having access to the Visual State product. See *Viewing design models via the Visual State Viewer*, page 907.

Table 3: IAR Visual State components overview

IAR Visual State also comes with:

Features for interoperability	Description
C-SPYLink	C-SPYLink allows you to perform high-level state machine debugging in the IAR Embedded Workbench C-SPY Debugger, which means that you can test on target hardware. See *Debugging design models using C-SPYLink*, page 759.
RealLink	Visual State RealLink is used with the Designer and allows you to test your state machine model on target hardware. See *Debugging design models using RealLink*, page 785.
Control Center	The Visual State Control Center provides third-party products with an interface to IAR Visual State. Among other things, the Control Center can be used for remote simulation of your state machine model. See *Using IAR Visual State remotely via the Control Center*, page 909.
Altia Design	Using Altia Design you can create a graphical interface prototype of your state machine model. Via the Validator you can connect your design model to Altia Design and simulate it. See *Briefly about prototyping with Altia Design*, page 884.
XMI file format	The XMI file format makes it possible to move design models between IAR Visual State and design tools from other vendors. See *Importing and exporting design models via XMI® files*, page 927.
State machine API	This open API with C-based access makes it possible to access your model from various programming languages. See *The Visual State State Machine API for programmatic manipulation of models*, page 931.

Table 4: IAR Visual State interoperability features

IAR Visual State filename extensions

These are the file types specific to IAR Visual State:

Filename extension	Description
bk<*x*>	Visual State Designer backup files.
cre	Visual State Coder report files.
stereotypes	Visual State Designer files for holding defined stereotypes.
vda	Visual State Validator dynamic analysis files.
vdg	Visual State Designer project diagram information (graphical animation).
vdi	Visual State Designer project diagram information.
vlg	Visual State Validator log files and animation files in the legacy format for sequence files.
vnw	Visual State Navigator workspace file.
vre	Visual State Verificator report files.
vsa	Visual State Validator static analysis files.
vsp	Visual State project files, which contain information about: * Visual State systems that make up the Visual State project * Visual State files that make up the Visual State systems * global element declarations.
vsr	Visual State state machine diagram files, which contain local element declarations and logic.
vsreqif	Visual State Coder requirements files.
vssm	Visual State state machine files.
vst	Visual State Designer interval backup files.
vste	Visual State transition element files.
vtg	Visual State project options files for the Coders, the Verificator, and the Documenter.
vws	Visual State Validator workspace files.
vxlg	Visual State sequence file
vws.bak	Visual State Validator workspace backup files.

Table 5: IAR Visual State filename extensions

IMPORTANT FEATURES AND ADVANTAGES

IAR Visual State provides many advantages and features.

Automatic code generation from a state machine model

Automatic generation of code for state machine models has a number of important advantages over manual code generation—first and foremost, state machine concurrency is taken care of automatically. Programming state machine concurrency tends to be complicated and error-prone.

The generated code makes no assumptions about any compiler-specific features except Standard C conformance, see *Standard C conformance*, page 460.

You can configure the Visual State Coders to use compiler-specific keywords to place state machine code and data in the memory areas of your choice. Size-of-data entities can be forced to 16 or 32 bits to match your target architecture for speed purposes, even if the model only requires 8-bit representation. You can configure the Coders in many different ways to balance the needs of the target MCU, the compiler, and coding standards.

The code generated by IAR Visual State focuses on the control logic of a state machine system. For several reasons, this part of the code should not be modified by hand, the most important reason being that the design is always the only explicit representation of the control logic. In that way, the model and the executing code always stay synchronized. Modifying state machine code by hand always carries the risk of introducing hard-to-find errors in the internal bookkeeping of states and conditions.

Product variant support in the model

Many products are available for the end user in multiple similar variants. There might be a Premium version and a Basic version of a product. For instance, much of the functionality of the product might be the same—but the Premium variant contains more features for the end user.

IAR Visual State supports defining product *variants* for situations where most of the feature set is identical but some features are different or only available for certain product variants. Using a shared base design avoids having to maintain two or more separate development tracks of the embedded software. See *Using variants and features*, page 217.

Simulation/validation of a state machine model

With design level simulation, you can start testing your model as soon as you have saved the first version of it. In this way you can find possible errors and omissions early in the development project, even before you have any hardware available.

Formal model checking of a state machine model

Formal verification helps you to identify possible problems in your code that are very hard to test for. A state might, for example, be impossible to exit after entering and exiting it a specific number of times, because of some blocking transition condition. If this was unintentional, it can be very difficult to find the problem using traditional testing methods.

Model debugging on target hardware

Debugging state machine code on C level is often difficult, because too many implementation details can obscure the design. With IAR Visual State you can debug on target hardware with feedback directly in the state machine diagram—to see exactly which state configuration is active and which transition was taken to enter that state configuration. You have these alternatives:

- C-SPYLink. If you use IAR Embedded Workbench® you can use the C-SPYLink plugin to pass high-level state machine model feedback directly to the IAR C-SPY® Debugger. C-SPYLink includes graphical animation in the state machine diagram when it executes, the possibility to set breakpoints at state machine level instead of C level, and trace and log functionality.
- RealLink. If you cannot use a hardware debug solution with the IAR C-SPY Debugger, you can use RealLink to communicate state machine data over a separate communication channel, for example, an RS232 port. RealLink can be used for any target that has a serial communication port where you can decide the communication speed, or if the target can use TCP/IP communication for debug purposes. RealLink is available if you make the Classic Coder generate C output with a table-based API.

See *Part 7. Testing your state machine model on hardware*, page 757 for more information about C-SPYLink and RealLink.

Support for high-integrity systems

IAR Visual State is suited for many design tasks that involve functional safety. For example, the IEC-61508 standard on functional safety explicitly recommends state machines as a design method to meet higher safety integrity levels.

You can use IAR Visual State's formal verification to find issues in your design that are almost impossible to fully cover with test suites. For example, you can find dead-end situations, unreachable parts of the design, never consumed input, etc. See *Part 5. Formal verification using the Verificator*, page 411.

UML (Unified Modeling Language)

IAR Visual State uses the UML notation for state machines. This notation is based on hierarchical state machines, and concurrency can be used at any level in the hierarchy. Variables are introduced and can be used as conditions, or be modified within the design. Actions can be used on transitions, and as entry and exit reactions.

IAR Visual State has been developed in accordance with the UML notation, but can also be used for designing state/event systems compliant with the Mealy notation.

For more information about the UML concepts, see the OMG Unified Modeling Language Specification, version 2.4.1, August 2011, available from **www.omg.org**.

Natural interrupt handling

The Visual State runtime execution engine deals with events—abstractions of occurrences in the environment. This makes it natural to map an interrupt to a Visual State event, provided that the interrupt affects the state machine.

If there is an event to process, a typical Visual State application runs the state machine engine as part of the main loop.

Exactly how the interrupt routine communicates with the state machine engine depends on your design. Implementation methods range from letting the interrupt routine set a flag that the main loop can detect, using a simple event queue with appropriate synchronization mechanism, to using a fully featured RTOS queue or semaphore.

The structure of your application is the same as usual. If an interrupt service routine generates input to the system of state machines, the routine simply puts the appropriate event into the state machine event queue and returns.

Asynchronous event handling

Asynchronous events are handled if they are forwarded to the Visual State engine. This is usually done by putting them into the event queue. As long as an event is in the event queue, it will eventually be processed by the Visual State control logic.

Easy integration with an RTOS

Use IAR Visual State to design the control logic of a task, or part of a task. Integrate your tasks with their respective priorities into the system with the RTOS just as if you were coding the application by hand.

To split Visual State code to run in different tasks, divide the state machines into different systems. A Visual State system is a collection of state machines that are designed as a unit, to run as a unit—possibly rather tightly coupled to each other. An RTOS application can contain any number of systems, and systems can communicate on task level using the available RTOS primitives.

Systems can be assigned arbitrarily to RTOS tasks, so that a task can actually contain more than one system at a time.

Prototyping a graphical interface for your model before having the hardware

You can easily integrate code generated by IAR Visual State with an application developed using a RAD tool like Altia Design, Microsoft® Visual C++®, or any other GUI toolchain of your choice.

For information about how to integrate with Altia Design, see *Briefly about prototyping with Altia Design*, page 884.

See also *Briefly about prototyping based on Coder-generated code*, page 887

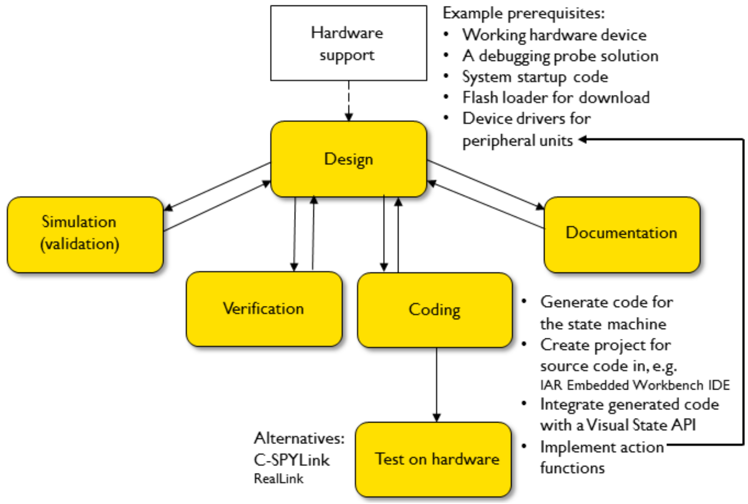
Application development using IAR Visual State

Learn more about:

- The application development cycle
- Control logic, data manipulation, and device drivers
- Code required for an application
- Project examples
- Sample source code

THE APPLICATION DEVELOPMENT CYCLE

This illustrates a typical development cycle using IAR Visual State:



Before you start using IAR Visual State, you should prepare your hardware setup. For example, it can be good to have a working hardware device, a debug probe solution, system startup code, and possibly also a flash loader for downloading your final application. However, none of this is mandatory before you start designing using IAR Visual State.

At some point during your development cycle, you must also write the source code for the device driver for the peripheral units.

This is the typical development cycle more in detail:

- Start the Navigator. This is where you set up your Visual State project in a workspace, including setting options for verification, code generation, and documentation. See *Part 2. Project management using the Navigator*, page 69
- Use the Designer to design your state machine models. See *Part 3. Designing using the Designer*, page 115.
- Use the Validator to simulate, validate, and debug the model. See *Part 4. Simulating using the Validator*, page 319. Typically, you iterate designing and simulation a couple of times.
- Use the Verificator to verify the logic of the model. See *Part 5. Formal verification using the Verificator*, page 411. Typically, this leads to redesigning some parts of your model.

- Coding involves several tasks:

  In the IDE of the compiler you are using (for example, the IAR Embedded Workbench IDE), create a project that includes all the necessary source code files.

  When you have tested your model in the Validator and corrected it in the Designer, you can generate the code for it. On target, the code will behave exactly as the model you designed. See *Part 6. Code generation using a Coder*, page 455.

  Integrate the generated code of your state machine model, using the Visual State API. See *Introduction to code generation, the Coders, and the APIs*, page 457.

  Implement the action functions for the peripheral units as required by your state machine model.
- Observe and control the runtime behavior of your models when they execute on hardware. For this you can use C-SPYLink or RealLink. See *Part 7. Testing your state machine model on hardware*, page 757.
- Use the Documenter to document your project, by creating a report. Typically, the report is useful for communicating the design with others. Of course, you can also do this very early during the initial design phase. See *Part 8. Documenting Visual State projects using the Documenter*, page 811.

CONTROL LOGIC, DATA MANIPULATION, AND DEVICE DRIVERS

A typical embedded application is a combination of code for control logic, data manipulation, and device drivers.

Device drivers for a specific target processor are usually written only once. You can make them part of a library, which remains more or less constant from project to project. Of course, the control logic part that implements the features and specification of a given product might change dramatically from project to project.

Using IAR Visual State, you develop the control logic for event-driven systems based on state machines, where events coming from external devices are processed by the control logic. Processing the events ultimately leads to actions on the environment. These actions will often interact with the device drivers for the hardware.

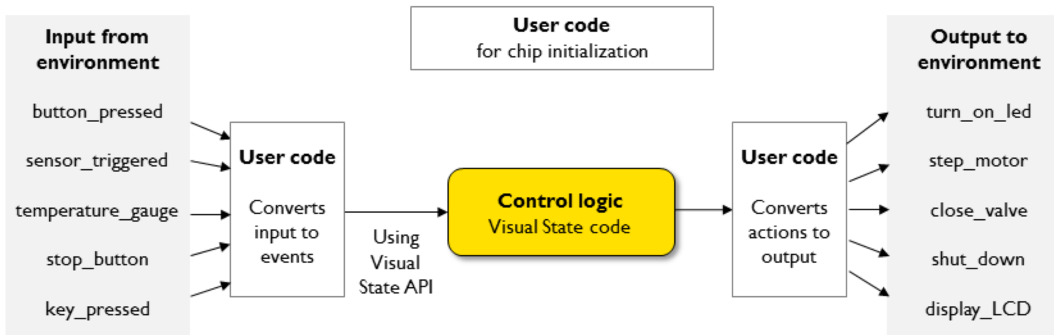
This is what happens:

1 The externally generated input is processed by the device driver, by way of interrupts or polling.

2 The driver informs the Visual State runtime execution engine, which acts according to the state machine model (changes states, executes actions, etc).

3 As a result of the state machine processing, actions (dedicated action functions) that use device drivers for output can be called.

This figure summarizes the parts that IAR Visual State handles in an embedded application:



CODE REQUIRED FOR AN APPLICATION

Creating an application using IAR Visual State as your main control logic engine is easy—you still have full control over the structure of your application code.

To create a final embedded application using Visual State-generated code, you must:

- Manually write code for event preprocessing (device drivers), event queues (if needed), and action functions (device drivers)
- Integrate the state machine in your application by calling its step function at the appropriate time. There are also optional facilities for inspection of the state machine.
- Integrate your code with the Coder-generated code, using a Visual State API.

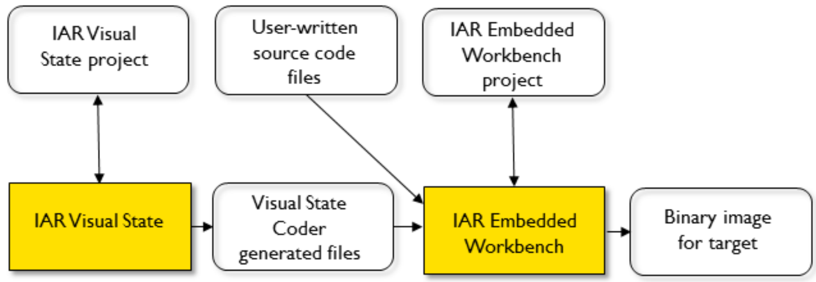
Action function invocations are automatically generated by IAR Visual State. However, you must write the code for each of the action functions.

Coder-generated code and the APIs

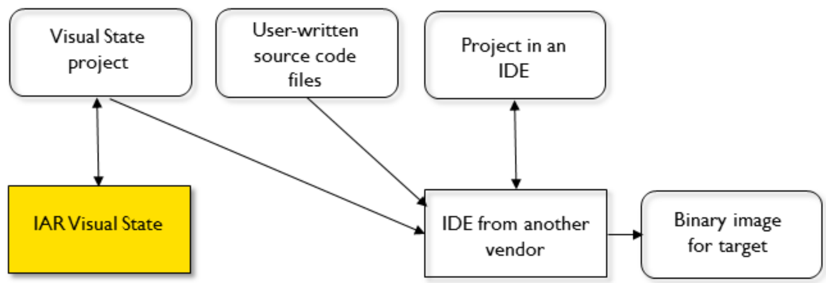
Coder-generated code is generated by the Visual State Coder from the state machine diagrams created in the Designer. The generated code must be integrated with your own user-written source code by means of a Visual State API.

Generating code in your Visual State project results in a number of source files. If you are using the IAR Embedded Workbench IDE, the generated source code files and dependency files are handled automatically if you include the project connection file (generated by IAR Visual State) in the IDE project. For information about how to include files in the IAR Embedded Workbench IDE project, see the *IDE Project Management and Building Guide*.

This figure shows an application development project using IAR Visual State and IAR Embedded Workbench:



This figure shows an application development project using IAR Visual State and an IDE from another vendor:



There are two types of Coder-generated code:

- Table-based code, which can be very compact
- Readable C code (requires the Visual State Classic Coder).

Which representation you choose depends on your specific application requirements regarding speed and size, and how important it is that you can examine the generated code manually.

For more information, see *Introduction to code generation, the Coders, and the APIs*, page 457.

PROJECT EXAMPLES

Your Visual State installation includes examples of application designs created with IAR Visual State. The examples can be useful for your own design as well as provide a reference for design techniques.

The examples can be opened from the Information Center in the Navigator.

SAMPLE SOURCE CODE

Your IAR Visual State installation includes sample source code that you can use as a source of reference in your development projects. The sample code files can be opened from the Information Center in the Navigator or via the `Examples` directory in the Visual State product installation.

Part 2. Project management using the Navigator

This part of the *IAR Visual State User Guide* includes these chapters:

- Project management
- The IAR Visual State Compare Tool
- Custom commands

Project management

- Introduction to project management using the Navigator
- Setting up workspaces and projects
- Graphical environment for the Navigator
- Reference information on Navigator menus.

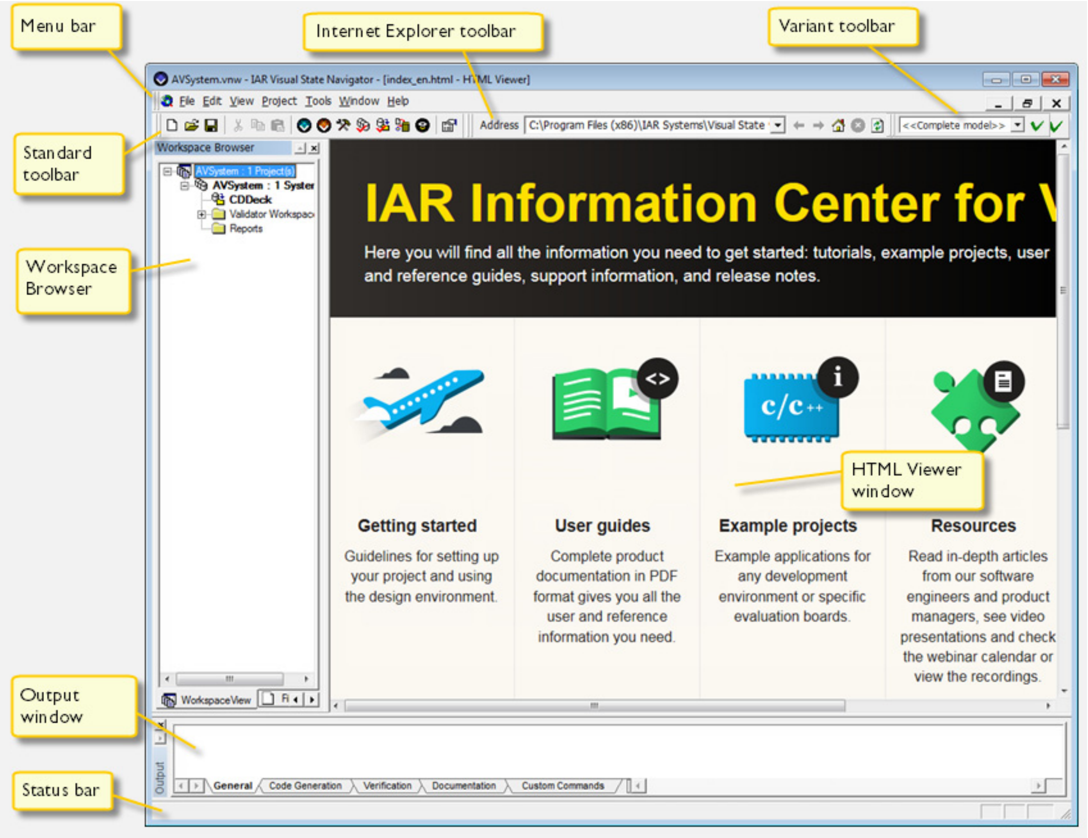
Introduction to project management using the Navigator

Learn more about:

- *Briefly about the Visual State Navigator*, page 71
- *The Visual State project*, page 72
- *The workspace*, page 73
- *Variants and features*, page 75

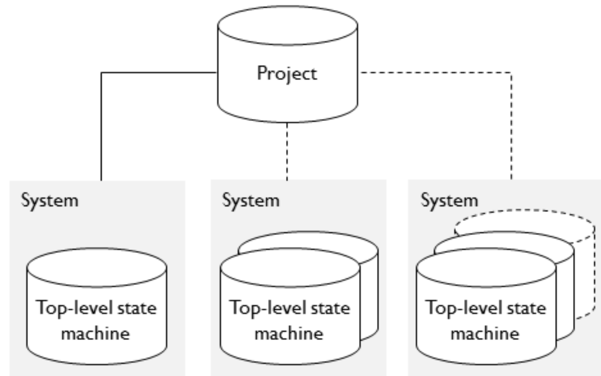
BRIEFLY ABOUT THE VISUAL STATE NAVIGATOR

The Visual State Navigator is a graphical project management tool for Visual State *projects*, from model design over test and simulation, to code generation and documentation. From the Navigator you access and activate the other components in IAR Visual State, and set options for the Verificator, Coder, and Documenter.



THE VISUAL STATE PROJECT

A Visual State *project* is a collection of Visual State *systems*. The systems group the individual *top-level state machines* together—in one file for each top-level state machine. Thus, each project can contain several systems, and each system can contain one or several top-level state machines:



In addition to binding systems and state machines together, a project contains elements that are shared across several systems. These elements are termed *transition elements*, see *Introduction to transition elements*, page 177. The project data is stored in a project file which has the filename extension `vsp`.

See also *Briefly about organizing your system*, page 124.

For more information about Visual State systems and state machines, see *Introduction to designing state machines using the Designer*, page 117.

THE WORKSPACE

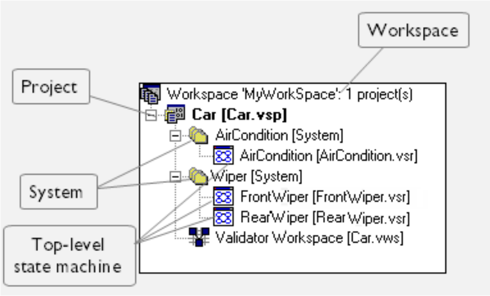
In the Navigator you set up your Visual State project in a *workspace*. The workspace organizes and handles one or several Visual State projects, systems, and state machine diagrams that are grouped together logically. The workspace contains links to Visual State projects, systems, and various types of files. The workspace is stored in a file with the filename extension `vnw`. A workspace file contains only one workspace.

In the workspace you can set options for verification, code generation, and project reports. The workspace can also be used for setting up your own commands.

You develop your Visual State project using the other Visual State components, available from the Navigator **Project** menu.

For more information, see *Setting up workspaces and projects*, page 75, *Setting Verificator, Coder, and Documenter options*, page 79, *Custom commands*, page 107.

This example shows a workspace and the structure of a project, viewed with the tree browser of the Navigator:



The project in this example is named `Car`. The project contains two systems named `AirCondition` and `Wiper`, respectively.

The generated file types are:

- The workspace file (`vnw`): can contain any number of projects
- The project file (`vsp`): can contain any number of systems
- The state machine file (`vsr`): can contain one top-level state machine

No file is generated for the system, which can contain any number of top-level state machines.

For more information, see *The Visual State system*, page 123.

Note: The Navigator workspace is not the same as the Validator workspace, a workspace used for testing. See *Part 4. Simulating using the Validator*, page 319.

Digital signatures for tracking inconsistencies

Each project has an associated digital signature, to track consistency between the files generated by the various components, and to track changes from version to version of a project.

The digital signature is a string value calculated from a project file and its associated state machine diagram files. Only the logical parts of the project are used in the calculation, not, for example, explanations to various elements. Every time a change is made to a project part that is used for calculating the signature, for example when an event is renamed, the digital signature also changes.

The digital signature is used in:

- Visual State code-generated files
- generated files included in the Documenter report

- the runtime application. The signature can be retrieved at runtime for diagnostic and other uses.

All files generated by the Validator, the Coders, and the Verificator that can be included in a report generated by the Documenter, will have a digital signature. By default, the Documenter will only include files with a valid digital signature. This behavior can be changed using the **File inclusion criteria** option, see *Documenter Options dialog box : File Input*, page 819.

See also *Digital signatures for tracking inconsistencies*, page 74.

VARIANTS AND FEATURES

If your product will be available for the end user in multiple similar variants, for example as a Premium version and a Basic version or as versions for different sales regions, IAR Visual State supports *variants* on a shared base design, to avoid having to maintain two or more separate software development tracks.

You specially mark up the parts of the model that should be excluded from the resulting code, and from testing. The result will contain the states, transitions, and transition elements in it that matches the setup for the variant in the Designer.

When you generate code, you choose one of the designed variants, or the complete model.

To enhance working with variants, the Designer also supports *features*, a subset of the design that can optionally be part of the model, or that is simply used to group features together. Each feature has a type that determines how you can include/exclude it in a variant meant for code generation.

You can use variants without using features. The features functionality is only needed if you have parts of the model that should be included in more than one of your runtime variants. However, combining features with variants gives you a very high flexibility to mark up areas in your design as functionality blocks to include/exclude for a variant.

For more information about variants and features, see *Using variants and features*, page 217.

Note: The use of features and variants is optional. All existing models will work as previously designed.

Setting up workspaces and projects

What do you want to do?

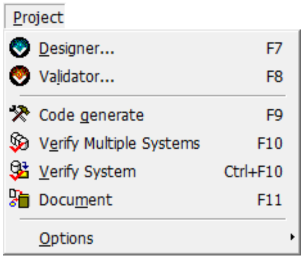
- *Starting IAR Visual State*, page 76
- *Creating a standard workspace*, page 76

- *Creating a new project in a workspace*, page 77
- *Adding an existing project to a workspace*, page 78
- *Setting a project or system as active*, page 79
- *Setting Verificator, Coder, and Documenter options*, page 79

STARTING IAR VISUAL STATE

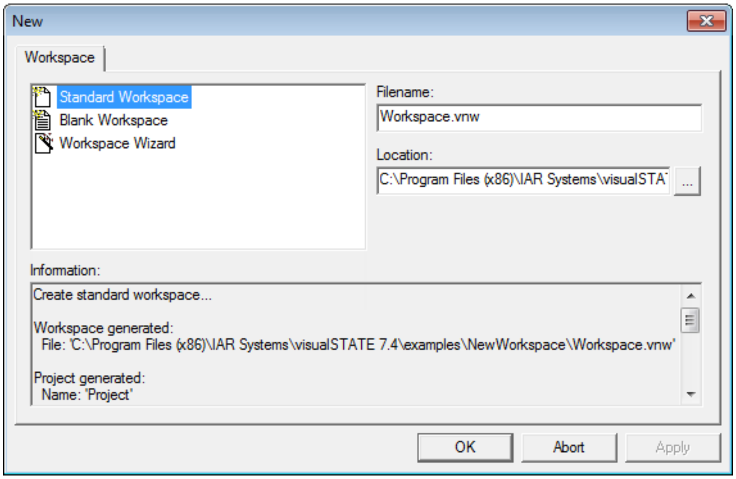
To start IAR Visual State, choose **Start menu>Programs>IAR Systems>Visual State**.

When you have created a workspace in the Navigator, you can start the other Visual State components and IAR Embedded Workbench using the buttons on the standard toolbar in the Navigator, commands from context menus, or from the Navigator **Project** menu:



CREATING A STANDARD WORKSPACE

1. In the Navigator, choose **File>New** to open a **New** dialog box:



2 On the **Workspace** page, select **Standard Workspace**. For reference information, see *New Workspace dialog box*, page 90.

In the **File name** and **Location** text fields, specify the filename and directory of your workspace file. Click **OK**.

A standard workspace is created, with a project that contains one system and one top-level state machine.

3 Now you can add a project to your workspace, see:

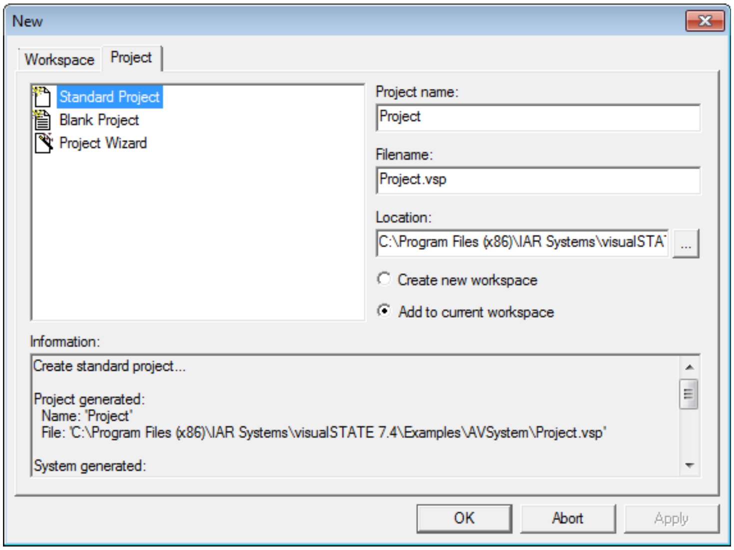
- *Creating a new project in a workspace*, page 77
- *Adding an existing project to a workspace*, page 78.

CREATING A NEW PROJECT IN A WORKSPACE

1 In the Navigator, choose **File>Open Workspace** to open your workspace.

2 Choose **File>New**.

3 In the **New** dialog box, click the **Project** tab:



4 Choose one of these alternatives:

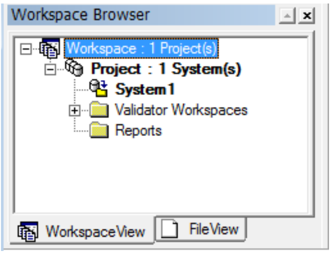
- **Standard Project**, to create a standard project with one system that contains one top-level state machine.

- **Blank Project**, to create an empty project where you can create your systems and top-level state machines.
- **Project Wizard**, to guide you through the process of creating a customized project, where you can specify the number of systems, top-level state machines, etc.

5 Specify a project name, a project filename (`vsp`), and the location of the project file.

6 Select **Add to current workspace** and click **OK**. The Designer is started.

7 Return to the Navigator. Click **Reload** to update the project if a message informs you that files have been modified outside of the application. The new project is inserted in the workspace:



Note: Selecting **Add to current workspace** will generate a workspace file with the same name as the project, and the filename 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 started with the project loaded.

8 Now you can set options, see *Setting Verificator, Coder, and Documenter options*, page 79.

ADDING AN EXISTING PROJECT TO A WORKSPACE

Projects created earlier or with the Designer can be added to a workspace.

1 In the Navigator, open your workspace.

2 Choose **File>Insert Project** to open the **Insert Visual State Project** dialog box.

3 Use the dialog box to locate the project and add it.

4 Now you can set options, see *Setting Verificator, Coder, and Documenter options*, page 79.

SETTING A PROJECT OR SYSTEM AS ACTIVE

You can set a project or system as *active*. This means that all operations you perform via the main menu will apply to that project or system. For example, **Project>Verify System** will verify the active system in the active project.

To set a project or system as active:

1. In the Navigator, open your workspace.
2. Select the system or project you want to set as active, right-click and choose **Set as Active Project/System**.

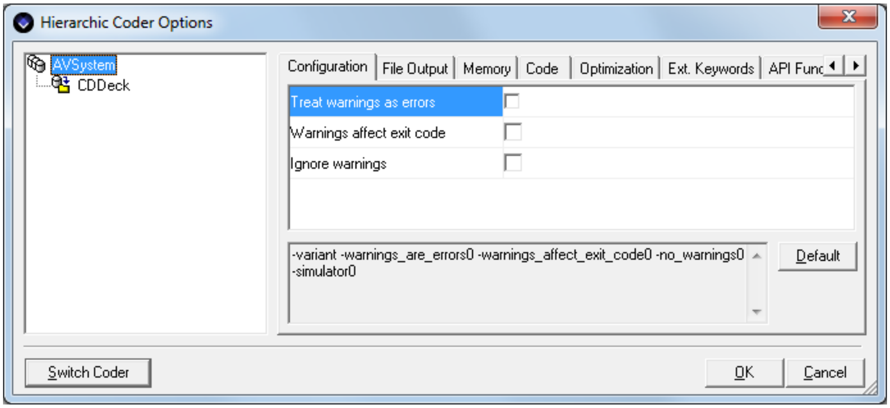
   The project or system you set as active will appear in bold in the **Workspace Browser** window.

   If you want to apply operations to a project or system that has not been set as active, select it in the Workspace view of the **Workspace Browser** window and use the commands on the context menu.

SETTING VERIFICATOR, CODER, AND DOCUMENTER OPTIONS

1. In the Navigator, open your workspace.
2. Choose one of these alternatives depending on which tool you want to set options for:
   - To set Verificator options, choose **Project>Options>Verification**.
     See *Verificator Options dialog box*, page 426 for reference information.
   - To set Coder options, choose **Project>Options>Code generation**.
     See *Classic Coder Options dialog box*, page 674 for reference information.
   - To set Documenter options, choose **Project>Options>Documentation**.
     See *Documenter Options dialog box*, page 816 for reference information.

3 An options dialog box is displayed:



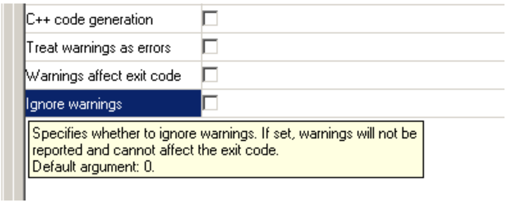
In this example, the **Coder Options** dialog box is used as an example. The dialog boxes **Verificator Options** and **Documenter Options** are used in the same way.

4 In the pane to the left, select the project or system you want to set options for.

5 Click the tab for the category of options you want to view. To view all available options, click the **All** tab.

Not all combinations of options are possible, so some options might be dimmed. Setting one option might disable another option. A different set of options might also be available on system level compared to project level.

Right-click an option or select the option and press Shift+F1 for detailed information on why it is unavailable. Alternatively, click F1 to get reference information about the dialog box an its options.



6 Make your settings, which will appear as command line options in the pane below.

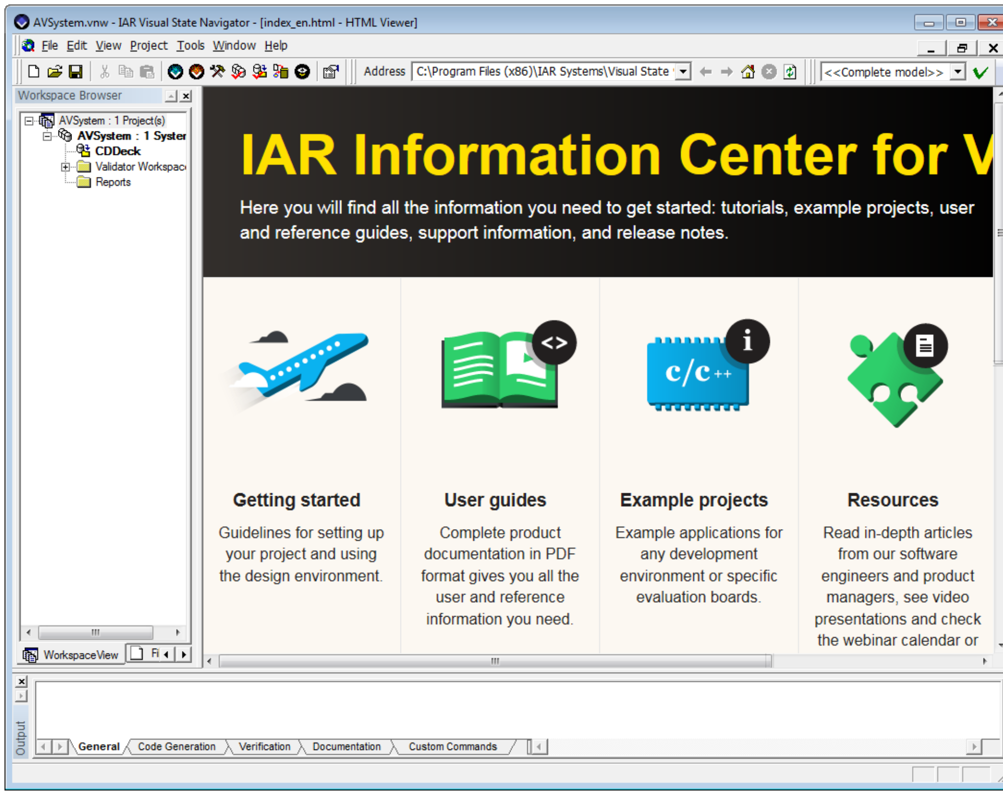
7 To restore the options to their default settings, click the **Default** button.

Graphical environment for the Navigator

Reference information about:

The Navigator main window

The main window of the Navigator is displayed when you start IAR Visual State.



The screenshot shows the window and its default layout.

The main window of the Navigator is a container for displaying the **Workspace Browser** window, an integrated **HTML Viewer** where the IAR Information Center appears by default, and the **Output** window. By default, all three windows are open.

Menu bar

The menu bar contains:

File

Commands for creating, opening, and saving workspaces and projects, printing, and exiting the Navigator. See *File menu*, page 94.

Edit

Standard Windows commands for working with text. See *Edit menu*, page 95.

View

Commands for opening windows and controlling which toolbars to display. See *View menu*, page 96.

Project

Commands for starting other Visual State components, for generating code, verifying, and documenting your project, and for setting options for the Visual State components. See *Project menu*, page 97.

Tools

Commands for starting IAR Embedded Workbench, for making Navigator settings, and for configuring custom commands. See *Tools menu*, page 98.

Window

Commands for changing how the Navigator windows are arranged on the screen. See *Window menu*, page 99.

Help

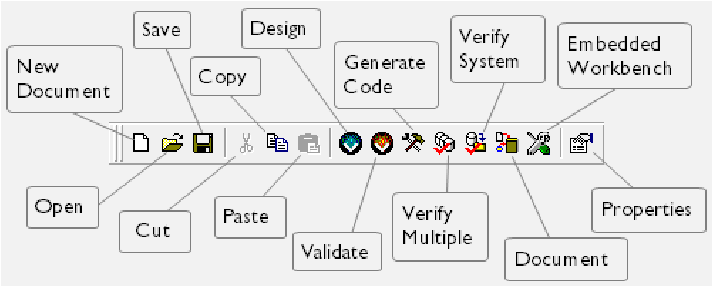
Commands that provide help about IAR Visual State. See *Help menu*, page 99.

Standard toolbar

The standard toolbar—available from the **View** menu—provides buttons for the most useful commands on the Navigator menus.

For a description of a button, point to it with the mouse pointer. The name of the button is displayed as a tooltip and a description is displayed in the status bar at the bottom of the main window. When a command is not available, the corresponding toolbar button is dimmed, and you will not be able to click it.

This figure shows the menu commands that correspond to each of the toolbar buttons:

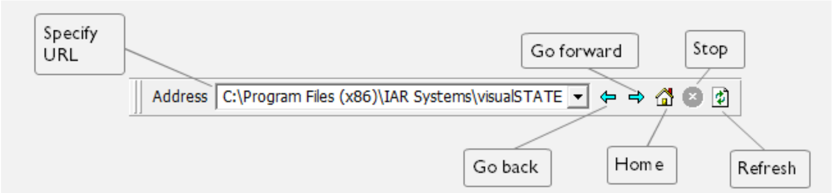


Internet browser toolbar

The Internet browser toolbar—available from the **View** menu—provides buttons for basic web browser commands.

For a description of a button, point to it with the mouse pointer. The name of the button is displayed as a tooltip and a description is displayed in the status bar at the bottom of the main window. When a command is not available, the corresponding toolbar button is dimmed, and you will not be able to click it.

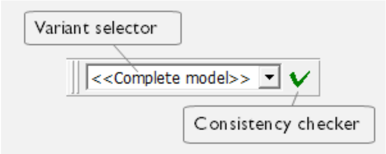
This figure shows the menu commands that correspond to each of the toolbar buttons:



Variant toolbar

The Variant toolbar—available from the **View** menu—controls the use of product variants in the model.

This figure shows the toolbar:



Variant selector

Choose which product *variant* that the Coder, Documenter, Verificator, and Validator operate on when they are called. If you choose **<<Complete model>>**, the Visual State components will operate on the entire model. The feature sets of the variants are edited inside the Designer.

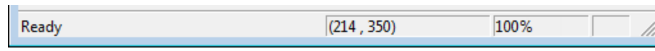
The active variant is saved as part of the Navigator workspace. For more information about variants, see *Using variants and features*, page 217.

Consistency checker

Performs a quick consistency check of the model before you open the model in the other Visual State components, restricting the model to the selected active variant. Errors are listed in the **Output** window. Use the Designer to correct any errors.

Status bar

The status bar at the bottom of the window can be enabled from the **View** menu.

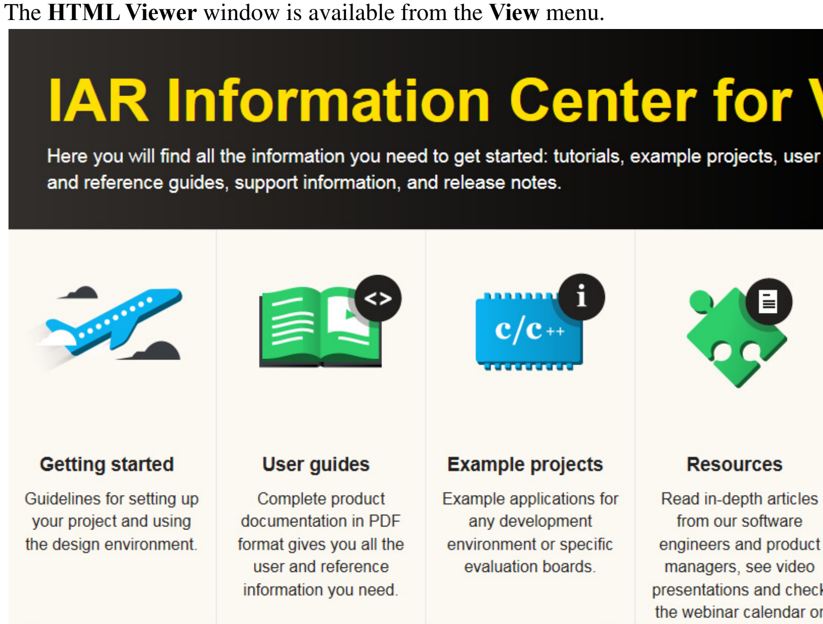


The status bar displays:

- The URL of links in the HTML Viewer window that you point to
- Descriptions of menu commands when you open a menu and hover over commands
- Descriptions of toolbar buttons that you point to
- The status of processes in the Navigator.

HTML Viewer window

The **HTML Viewer** window is available from the **View** menu.



This window is an integrated web browser, using your installed copy of Internet Explorer.

You can use the **Internet browser** toolbar to browse for other HTML pages.

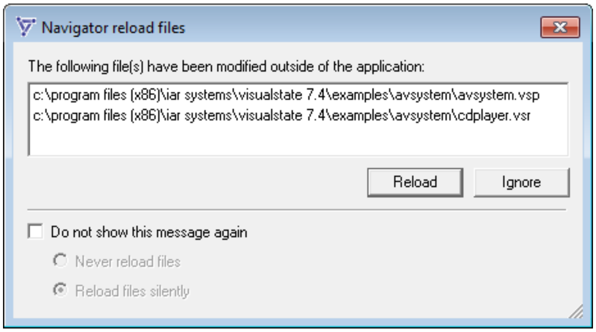
When you start IAR Visual State, the Navigator **HTML Viewer** window shows the IAR Information Center.

Context menu

The standard Internet Explorer context menu is available in the **HTML Viewer** window. For information about the commands, see the documentation from Microsoft Corporation.

Navigator Reload Files dialog box

The **Navigator Reload Files** dialog box is displayed when project files or state machine diagram files (`vsp` and `vsr` files) in the current workspace have been modified outside the Navigator.



The following file(s) have been modified outside the application

Lists the files that have been modified outside the Navigator.

Reload

Updates the information for all modified projects and systems in the **Workspace Browser** window.

Ignore

Closes the dialog box without updating the information for modified projects and systems in the **Workspace Browser** window.

Do not show this message again

This option disables the reloading message dialog box and either disables reloading or makes reloading automatic. Choose between:

Never reload files

Projects, systems, and state machine diagrams are only reloaded when the workspace is opened. This setting is not recommended.

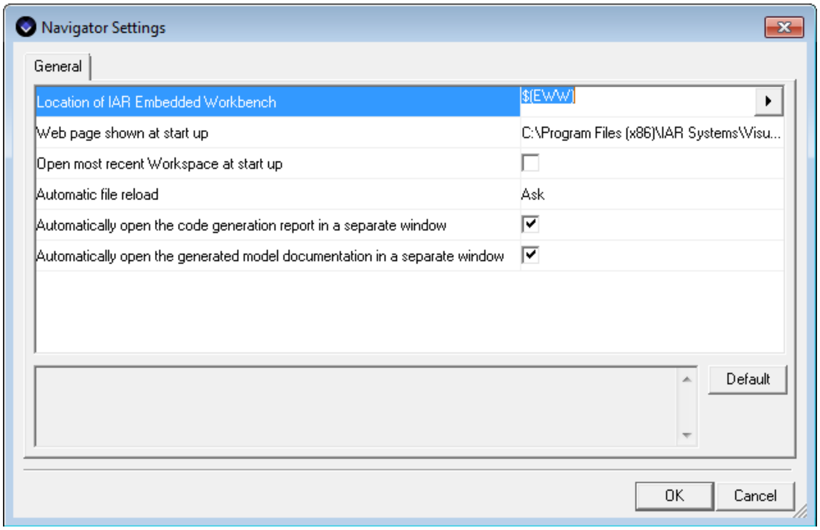
Reload files silently

All relevant projects, systems, and state machine diagrams are reloaded automatically when they have been changed outside the Navigator.

To re-enable the reloading message, use the option **Automatic file reload**, see *Navigator Settings dialog box*, page 87.

Navigator Settings dialog box

The **Navigator Settings** dialog box is available from the **Tools** menu.



Use this dialog box to make settings for the Navigator.

Location of IAR Embedded Workbench

Specify in which directory the IAR Embedded Workbench program is located. If this field is empty, IAR Embedded Workbench cannot be started from the **Tools** menu or the Navigator toolbar.

Web page shown at startup

Specify an HTML or plain text file to display when the Navigator starts and when you click the **Home** button on the Internet browser toolbar.

Open most recent workspace at startup

Determines whether the most recently used workspace is opened automatically when the Navigator starts.

Automatic file reload

Controls whether projects, systems, and state machine diagrams are reloaded when they have changed outside the Navigator. This updates the graphical information in the **Workspace Browser** window, and information is written to the **General** page of the **Output** window. Choose between:

Never

Projects, systems, and state machine diagrams are only reloaded when the workspace is opened. This setting is not recommended.

Ask

A dialog box asks whether you want to reload the files when a project, system, or state machine diagrams has changed outside the Navigator.

Always

All relevant projects, systems, and state machine diagrams are reloaded automatically when they have been changed outside the Navigator.

Automatically open the code generation report in a separate window

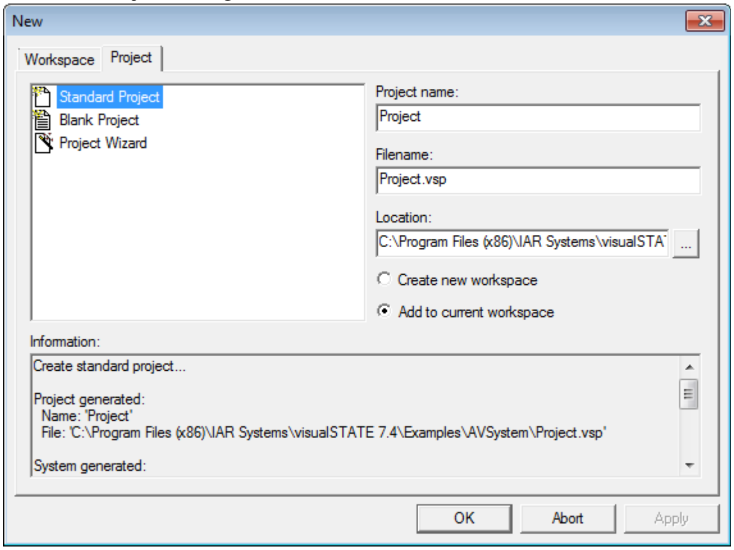
Determines whether the report is opened automatically every time you have generated code with the Coder.

Automatically open the generated model documentation in a separate window

Determines whether the model documentation is opened automatically every time you have generated documentation with the Documenter.

New Project dialog box

The **New Project** dialog box is available from the **File** menu.



Use this dialog box to create a new project.

See also *Creating a new project in a workspace*, page 77.

Display area

Select the type of project you want to create:

Standard Project

Creates a standard project with one system, that contains one top-level state machine.

Blank Project

Creates an empty project without systems.

Project Wizard

Opens the Project wizard, to guide you through the process of creating a customized project.

Project name

Type a name for the project you are creating.

Filename

Type a name for the project file in which the project will be stored.

Location

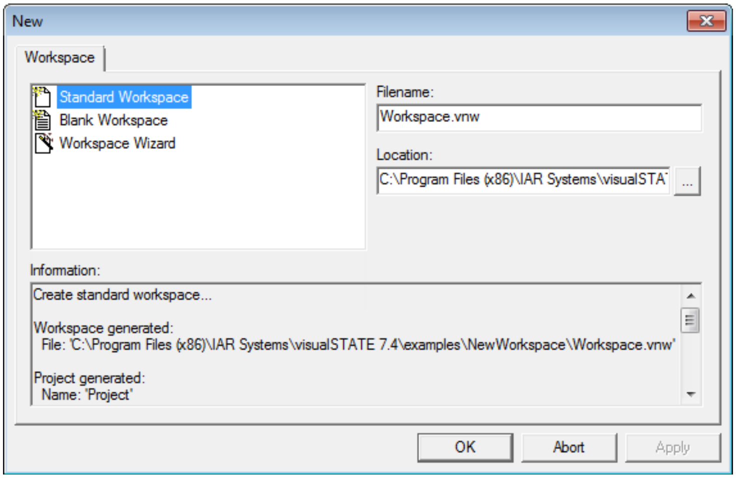
Browse to the directory where you want to create the new project file.

Information

Describes what the result of your actions in this dialog box will be when you click **OK**.

New Workspace dialog box

The **New Workspace** dialog box is available from the **File** menu.



Use this dialog box to create a new workspace.

See also *Creating a standard workspace*, page 76.

Display area

Select the type of workspace you want to create:

Standard Workspace

Creates a standard workspace with one project, containing one top-level state machine.

Blank Workspace

Creates an blank workspace without projects or systems.

Workspace Wizard

Opens the Workspace wizard, to guide you through the process of creating a workspace.

Filename

Type a name for the workspace file you are creating.

Location

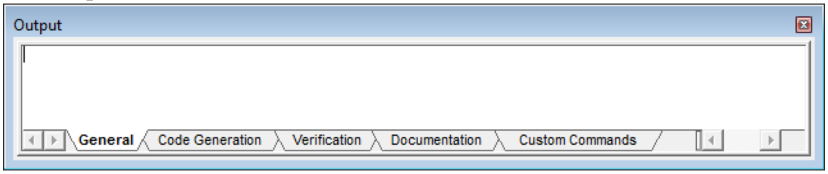
Browse to the directory where you want to create the new workspace file.

Information

Describes what the result of your actions in this dialog box will be when you click **OK**.

Output window

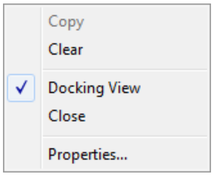
The **Output** window is available from the **View** menu.



This window displays information about the loaded workspace. The tabbed pages contain general information from the Verificator, Coder, and Documenter when these components are running.

Context menu

This context menu is available:



These commands are available:

Copy

Copies the selected text in the window.

Clear

Deletes all text for the active view in the window.

Docking View

Toggles between docking the window in the Navigator main window and making it float.

Close

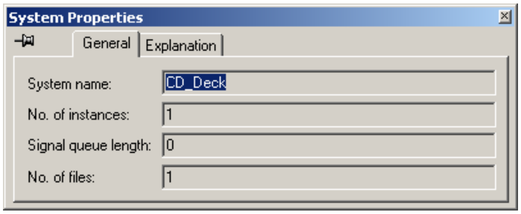
Closes the window.

Properties

Opens the **Properties** window for the active view in the window. See *Properties window*, page 92.

Properties window

The **Properties** window is available from the **View** menu and from the context menu of the various Navigator windows.

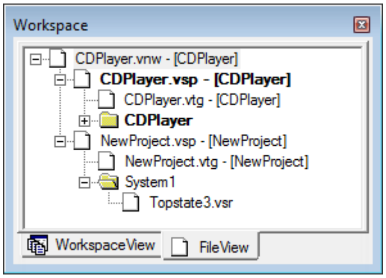


This window shows information about the currently selected or active window or object.

To make the window remain on the screen, click the pin icon .

Workspace Browser window

The **Workspace Browser** window is available from the **View** menu.



This window contains a browser where you can see the structure of the loaded workspace.

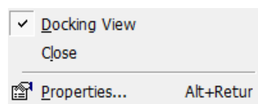
The window has two different views:

- A file view which shows the file structure of the workspace file, with project files, state machine diagram files, and system folders.
- A workspace view which shows the model structure of the projects in the workspace. This view also shows project-related items such as Validator workspaces and custom commands.

For more information about the workspace, see *The workspace*, page 73.

Context menu

This context menu is available:



These commands are available:

Docking View

Toggles between docking the window in the Navigator main window and making it float.

Close

Closes the window.

Properties

Opens the **Properties** window for the selected item in the browser. See *Properties window*, page 92.

Reference information on Navigator menus

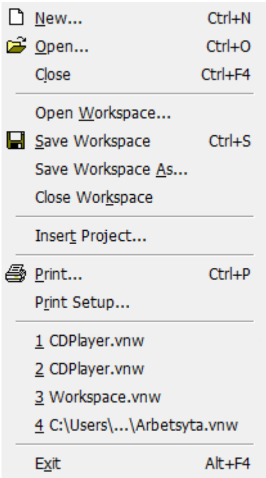
Reference information about:

- *File menu*, page 94
- *Edit menu*, page 95
- *View menu*, page 96
- *Project menu*, page 97
- *Tools menu*, page 98
- *Window menu*, page 99
- *Help menu*, page 99
- *Navigator shortcut key summary*, page 100.

File menu

The **File** menu provides commands for creating or opening workspaces, projects and web files, saving and printing, and exiting IAR Visual State.

The menu also includes a numbered list of the most recently opened workspaces. To open one of them, choose it from the menu.



Menu commands

These commands are available on the menu:

New (Ctrl+N)

Displays a dialog box where you can create a new workspace or project. See *New Workspace dialog box*, page 90 and *New Project dialog box*, page 89.

Open (Ctrl+O)

Displays a standard dialog box where you can open a workspace or web document.

Close (Ctrl+F4)

Closes the active document in the HTML browser window.

Open Workspace

Displays a standard dialog box where you can open a workspace.

Save Workspace (Ctrl+S)

Saves the current workspace file.

Save Workspace As

Displays a standard dialog box where you can save the current workspace file with a new name.

Close Workspace

Closes the current workspace. You will be given the opportunity to save any files that have been modified before closing.

Insert Project

Displays a standard dialog box where you can locate a project to add to the workspace.

Print (Ctrl+P)

Displays a dialog box where you can print the active document in the HTML browser window.

Print Setup

Displays a dialog box where you can set printer options.

***filename*.vnw**

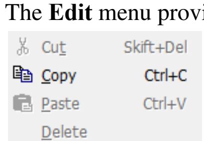
A numbered list of the most recently opened workspace files, in reverse order of when they were last opened. Choose the one you want to open.

Exit

Exits the Navigator. You will be asked whether to save any changes to files before they are closed.

Edit menu

The **Edit** menu provides commands for editing.



Menu commands

These commands are available on the menu:

Cut (Shift+Del)

The standard Windows command for copying text. This command is not available on the **Edit** menu in the Navigator, only as a shortcut key and from context menus.

Copy (Ctrl+C)

The standard Windows command for copying text.

Paste (Ctrl+V)

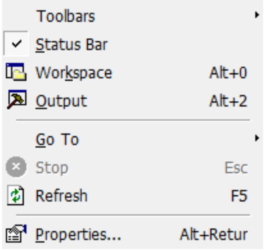
The standard Windows command for pasting text. This command is not available on the **Edit** menu in the Navigator, only as a shortcut key and from context menus.

Delete

The standard Windows command for deleting text.This command is not available on the **Edit** menu in the Navigator, only as a shortcut key and from context menus.

View menu

The **View** menu provides commands for opening windows and displaying toolbars.



Menu commands

These commands are available on the menu:

Toolbars

Opens a submenu, where the commands **Standard Bar**, **Internet Browser Bar**, and **Variant Bar** show/hide the three toolbars.

Status Bar

Shows/hides the status bar.

Workspace Browser (Alt+0)

Opens the current **Workspace Browser** window, see *Workspace Browser window*, page 92.

Output (Alt+2)

Opens the **Output** window, which contains information about the workspace and information from the Verificator, Coder, and Documenter. See *Output window*, page 91.

Go to>Back (Alt+Left Arrow)

The standard Internet Explorer command for going to the previous page in the browsing history in the current HTML browser window.

Go to>Forward (Alt+Right Arrow)

The standard Internet Explorer command for going to the next page in the browsing history in the current HTML browser window.

Go to>Home Page (Alt+Home)

The standard Internet Explorer command for going to the HTML browser home page.

Go to>URL

Sets the **Specify URL** field of the **Internet Browser** toolbar in focus.

Stop (Esc)

Stops the loading of the current HTML browser window.

Refresh (F5)

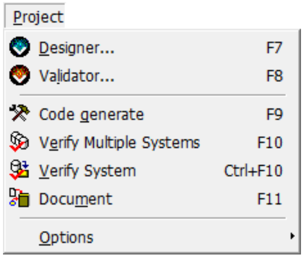
Reloads the contents of the current HTML browser window.

Properties (Alt+Enter)

Displays information about the current HTML browser window.

Project menu

The **Project** menu provides commands for starting the various Visual State components.



Menu commands

These commands are available on the menu:

Designer (F7)

Starts the Designer with the current project loaded. See *Part 3. Designing using the Designer*, page 115.

Validator (F8)

Starts the Validator with the current project loaded. See *Part 4. Simulating using the Validator*, page 319.

Code Generate (F9)

Starts code generation for the selected project. See *Part 6. Code generation using a Coder*, page 455.

Verify Multiple Systems (F10)

Verifies one or more systems in the selected project. See *Part 5. Formal verification using the Verificator*, page 411.

Verify System (Ctrl+F10)

Verifies the selected system. See *Part 5. Formal verification using the Verificator*, page 411.

Document (F11)

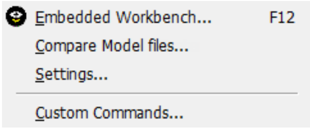
Creates a documentation report for the selected project. See *Part 8. Documenting Visual State projects using the Documenter*, page 811

Options (Alt+F9, Alt+F10, Alt+F11)

Opens a submenu where you can open the options dialog box for setting Coder, Verificator, and Documenter options. See *Classic Coder Options dialog box*, page 674, *Verificator Options dialog box*, page 426, and *Documenter Options dialog box*, page 816, respectively.

Tools menu

The **Tools** menu provides commands for starting IAR Embedded Workbench and for making Navigator settings.



Menu commands

These commands are available on the menu:

Embedded Workbench (F12)

Starts IAR Embedded Workbench, if it is installed and you have specified its location using the **Location of IAR Embedded Workbench** option in the **Navigator Settings** dialog box. See *Navigator Settings dialog box*, page 87. If there are more than one IAR Embedded Workbench product installed, the one specified in the dialog box will be started.

Compare Model Files

Launches the IAR Visual State Compare Tool, see *The IAR Visual State Compare Tool*, page 101.

Settings

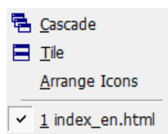
Displays the **Navigator Settings** dialog box, see *Navigator Settings dialog box*, page 87.

Custom Commands

Displays the **Custom Commands** dialog box, see *Custom Commands dialog box*, page 111. See also *Custom commands*, page 107.

Window menu

The **Window** menu provides commands for arranging the Navigator windows.



Menu commands

These commands are available on the menu:

Cascade

Arranges the open HTML browser windows partially on top of each other but fanned out so that the window titles are visible.

Tile

Changes the size of the open HTML browser windows and arranges them side by side so that they are all visible.

Arrange Icons

Arranges iconized windows.

Window name

A numbered list of the open HTML browser windows, in order of when they were opened. Choose the one you want to shift focus to.

Help menu

The **Help** menu provides help for IAR Visual State and displays the version number of the Navigator.

You can also access the Information Center from the **Help** menu. The Information Center is an integrated navigation system that gives easy access to the information resources you need to get started and during your project development: tutorials, example projects, user guides, support information, and release notes. It also provides shortcuts to useful sections on the IAR Systems web site.

Navigator shortcut key summary

These are the shortcut keys:

Description	Shortcut key
Create a new workspace, project, system, or state machine file	Ctrl+N
Open an existing file	Ctrl+O
Save the active window	Ctrl+S
Close the active window	Ctrl+F4
Print the active window	Ctrl+P
Make the **Workspace Browser** window the active window	Alt+0
Make the **Output** window the active window	Alt+2
Go to the previous page in the browsing history in the active HTML browser window	Alt+Left Arrow
Go to the next page in the browsing history in the active HTML browser window	Alt+Right Arrow
Go to the HTML browser home page.	Alt+Home
Stop loading the current HTML browser window	Esc
Reload the active window	F5
Show information about the current HTML browser window	Alt+Enter
Open the help system (context-sensitive)	F1
Close the active window	Alt+F4
Start the Designer with the current project loaded	F7
Start the Validator with the current project loaded	F8
Start code generation for the selected project	F9
Verify one or more systems in the selected project	F10
Verify the selected system	Ctrl+F10
Create a documentation report for the selected project	F11
Start IAR Embedded Workbench	F12
Display the **Coder Options** dialog box	Alt+F9
Display the **Verificator Options** dialog box	Alt+F10
Display the **Documenter Options** dialog box	Alt+F11

Table 6: Navigator shortcut keys

The IAR Visual State Compare Tool

- Introduction to the IAR Visual State Compare Tool
- Using the IAR Visual State Compare Tool
- Reference information on the IAR Visual State Compare Tool

Introduction to the IAR Visual State Compare Tool

The IAR Visual State Compare Tool can be used for visualizing differences between two state machine models or two complete projects. Using this tool is a complement to using a traditional text file comparison tool directly on the XML files.

These file types can be compared:

- `.vsp` (Visual State project files)
- `.vsr` (Visual State state machine diagram files)
- `.vssm` (Visual State state machine files)
- `.vste` (Visual State transition element files)

Using the IAR Visual State Compare Tool

To compare two Visual State files:

1. In the Navigator, choose **Tools>Compare Model Files** to open the IAR Visual State Compare Tool.
2. Click the browse button for **File A (base)** and navigate to the older version of the file and load it.
3. Click the browse button for **File B** and navigate to the newer version of the file and load it.
4. View the differences between the two versions, under the heading **Differences**. The information will normally load automatically. (If you change the name of a file, or if you make changes to the contents of a model, you might have to refresh the information by choosing **Commands>Compare Files**.)

5 To inspect a change, double-click the description in the list or right-click on it and choose **Show in Designer** from the context menu. This will open two Designer windows, one for each version of the state machine model/project, with the changed element selected. If the difference is not a diagram element, a standard Find operation is launched in the Designer window, see *Searching for a transition element*, page 193.

Note: If you loaded the files in the wrong order, or if you want to view the changes from a reverse perspective (additions as deletions, etc), choose **Commands>Switch A and B**.

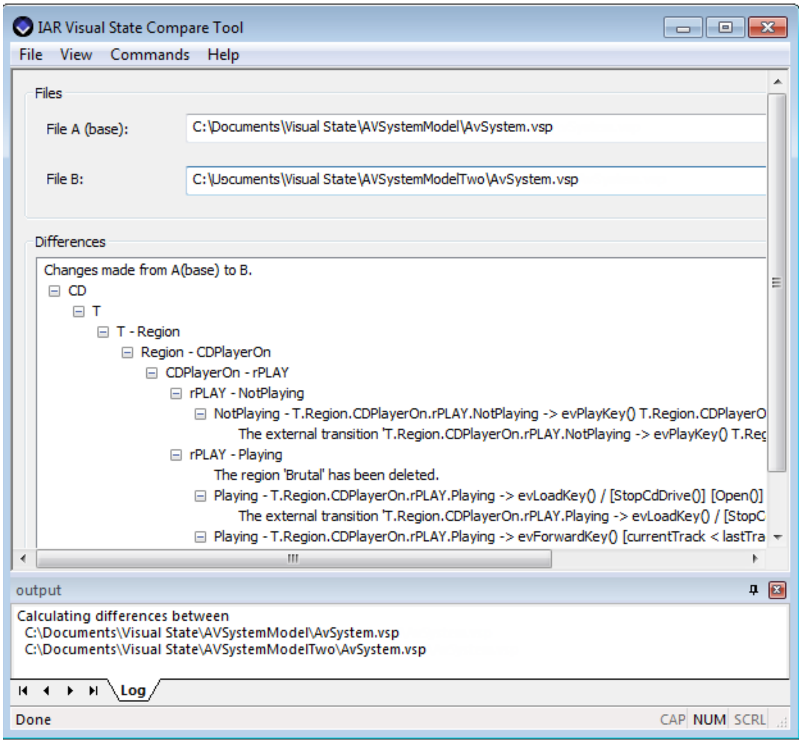
Reference information on the IAR Visual State Compare Tool

Reference information about:

- *IAR Visual State Compare Tool window*, page 103
- *File menu*, page 104
- *View menu*, page 104
- *Commands menu*, page 105
- *Help menu*, page 105

IAR Visual State Compare Tool window

The **IAR Visual State Compare Tool** window is available from the **Tools** menu in the Navigator.



This window displays detailed descriptions of all changes to a state machine model or project.

Files

Use the browse buttons to load the two versions of the file that you want to compare:

File A (base)

This field contains the path to the older version of the file that you want to compare.

File B

This field contains the path to the newer version of the file that you want to compare.

Differences

This area displays a detailed list of all differences between the two versions of your state machine file or project. If the differences are too fundamental, the IAR Visual State Compare Tool will conclude that they are not versions of the same file, and no comparison will be made.

To inspect a change, double-click the description in the list or right-click on it and choose **Show in Designer** from the context menu.

Output

The log in the **output** area displays a detailed list of all commands you send to the IAR Visual State Compare Tool. To clear the log, right-click in this area and choose **Clear** from the context menu.

File menu

The **File** menu provides commands for exiting the IAR Visual State Compare Tool.

Menu commands

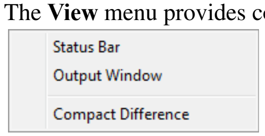
These commands are available on the menu:

Exit

Exits the IAR Visual State Compare Tool.

View menu

The **View** menu provides commands for displaying contents.



Menu commands

These commands are available on the menu:

Status Bar

Displays or hides the status bar at the bottom of the window.

Output Window

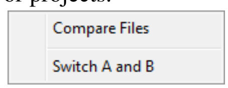
Displays or hides the output area with the log.

Compact Difference

Toggles the description of the changes in the **Differences** area between a detailed hierarchical view and a compact flat list.

Commands menu

The **Commands** menu provides commands for comparing state machine diagram files or projects.



Menu commands

These commands are available on the menu:

Compare Files

Refreshes the contents of the **Differences** area. The information will normally load automatically, but if you change the name of a file, or if you make changes to the contents of a model, you might have to refresh the comparison.

Switch A and B

Reverses the comparison to go the other way and refreshes the contents of the **Differences** area.

Help menu

The **Help** menu provides commands for displaying information about the IAR Visual State Compare Tool.

Custom commands

- Introduction to custom commands
- Using custom commands
- Graphical environment for custom commands

Introduction to custom commands

Learn more about:

- *Briefly about custom commands*, page 107

BRIEFLY ABOUT CUSTOM COMMANDS

You can defined a custom command to perform a specific task, for example compiling an entire Visual State 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 were created and to its projects.

Using custom commands

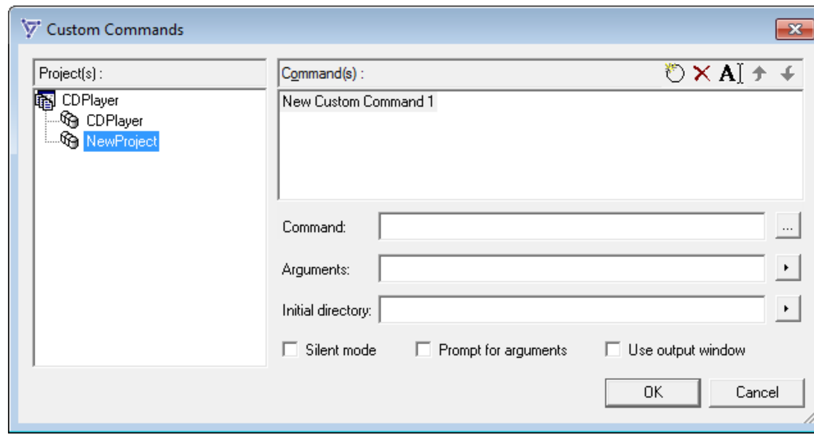
What do you want to do?

- Creating a custom command
- Executing a custom command
- Editing or deleting a custom command
- Renumbering custom command macros

CREATING A CUSTOM COMMAND

1 Start the Navigator and open your workspace file.

2 Choose **Tools>Custom Commands** to display the **Custom Commands** dialog box:

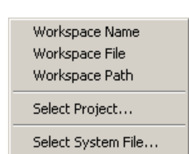


3 In the **Project(s)** pane, select the workspace or a project, depending on whether you want to create a workspace-specific command or a project-specific command. Workspace-specific commands can operate on all projects in the entire workspace. Project-specific custom commands only have access to the project for which they are defined.

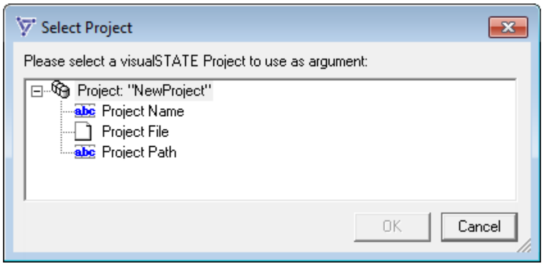
4 On the **Command(s)** toolbar, click the **New** button . Click the name and specify a more descriptive name.

5 In the **Command** field, specify the path to the program you want to be executed. There is a browse button available for your convenience.

6 In the **Arguments** field, type the arguments to be used by the custom command or click the  button to display a menu of arguments to choose from:

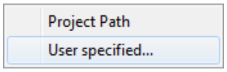


Choosing **Select Project** or **Select System File** displays a dialog box for selecting the item you want to use as an argument:



For example, selecting **Project File** inserts the macro `$(P0_FILE)`. When the custom command is activated, `$(P0_FILE)` is expanded to the name of the first project file in the workspace.

7 In the **Initial directory** field, type the directory to change to during execution of the custom command or click the button to display a menu of locations to choose from:



Choosing **User-specified** displays a dialog box for browsing to the initial directory you want to use.

8 Click **OK** to add the new custom command to the **Project Custom Commands** folder in the **Workspace Browser** window. Save the workspace.

For information about the options **Silent mode**, **Prompt for arguments**, and **Use output window**, see *Custom Commands dialog box*, page 111.

EXECUTING A CUSTOM COMMAND

1 Start the Navigator and open your workspace file.

2 In the **Workspace Browser** window, double-click the custom command you want to execute.

EDITING OR DELETING A CUSTOM COMMAND

1 Start the Navigator and open your workspace file.

2 Choose **Tools>Custom Commands** to display the **Custom Commands** dialog box.

3 In the **Command(s)** list, select the custom command you want to edit or delete.

4 Choose what you want to do:

- To edit the command, change the settings below the **Command(s)** list and click **OK**.
- To rename the command, click the **Rename** button on the toolbar and edit the name in the list.
- To delete the command, click the **Delete** button on the toolbar.

RENUMBERING CUSTOM COMMAND MACROS

The macros for your custom commands refer to projects and systems by number. To ensure that they will refer to the correct system if you have removed or added any systems, you must update the numbering.

To renumber your custom command macros:

1 Start the Navigator and open your workspace file.

2 Choose **Tools>Settings** to open the **Navigator Settings** dialog box.

3 Change the setting for the option **Renumbering of custom command macros**, see *Navigator Settings dialog box*, page 87.

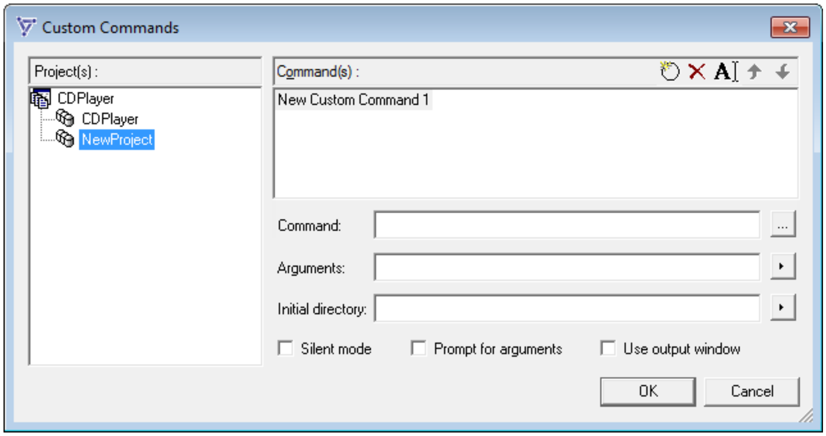
Graphical environment for custom commands

Reference information about:

- *Custom Commands dialog box*, page 111

Custom Commands dialog box

The **Custom Commands** dialog box is available from the **Tools** menu.



Use this dialog box to create or edit custom commands.

See also *Creating a custom command*, page 107.

Project(s)

Displays the projects of the loaded Visual State workspace.

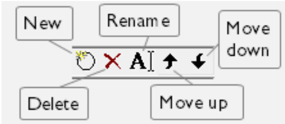
Command(s)

A list of custom commands that have been created for the workspace or project that is selected in the **Project(s)** pane.

Toolbar

The toolbar provides buttons for editing and manipulating the custom commands.

This figure shows the operations that correspond to each of the toolbar buttons:



New

Creates a new custom command.

Delete

Deletes the selected custom command.

Rename

Makes the name of the selected custom command editable.

Move Up

Moves the selected custom command upward in the list.

Move Down

Moves the selected custom command downward in the list.

Command

Specify the path to the program you want to be executed. There is a browse button available for your convenience.

Arguments

Type the arguments to be used by the custom command or click the button to display a menu of arguments to choose from. Choose between:

Workspace Name

Inserts the macro `$(WS_NAME)`. When the custom command is executed, this macro expands to the name of the workspace.

Workspace File

Inserts the macro `$(WS_FILE)`. When the custom command is executed, this macro expands to the name of the workspace file.

Workspace Path

Inserts the macro `$(WS_PATH)`. When the custom command is executed, this macro expands to the path of the workspace file.

Select Project

Displays a dialog box for selecting the project name, project filename, or project path that you want to use as an argument. This will insert one of the macros `$(PO_NAME)`, `$(PO_FILE)`, or `$(PO_PATH)`.

Select System File

Displays a dialog box for selecting the system name that you want to use as an argument. This will insert the macro `$(PO_SO_NAME)`.

Initial directory

Type the directory to change to during execution of the custom command or click the ▸ button to display a menu of locations to choose from. Choose between:

Project Path

Inserts the macro `$(PO_PATH)`. When the custom command is activated, this macro expands to the path of the project file.

User-specified

Displays a dialog box for browsing to the initial directory you want to use.

Silent mode

Makes the execution of the custom command be performed without displaying any windows or dialog boxes.

Note: This disables also any dialog boxes that request user interaction.

Prompt for arguments

Prompts you for arguments, if needed, during the execution of the custom command.

Use output window

Prints any output from the execution of the custom command to the **Custom Command** page of the Navigator **Output** window.

Part 3. Designing using the Designer

This part of the *IAR Visual State User Guide* includes these chapters:

- Designing
- States
- Transitions
- Transition elements
- Reusing designs using state machine templates
- Using variants and features
- Using requirements files
- The Visual State Designer

Designing

- Introduction to designing state machines using the Designer
- Designing state machines

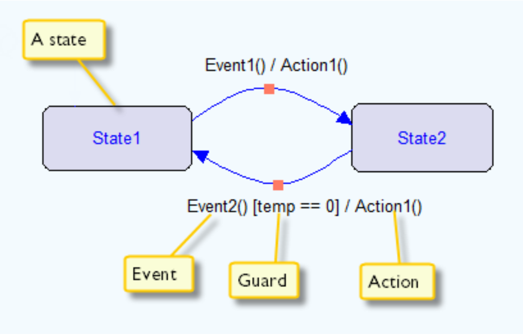
Introduction to designing state machines using the Designer

Learn more about:

BRIEFLY ABOUT STATE MACHINES AND DESIGNING

State machines are commonly used for describing discrete systems, where the current behavior is a result of previously occurring events. A state machine transforms incoming events to deduced out-going actions, the machine is just a reactive engine or core, not to be confused with an operating system.

A state machine consists of a finite set of *states* in a hierarchy and a collection of *transitions*. The states represent the possible situations in the system, and the transitions represent a change from one state to another. The system can change states depending on input from the environment (events). As a state change occurs, actions can be performed on the environment:



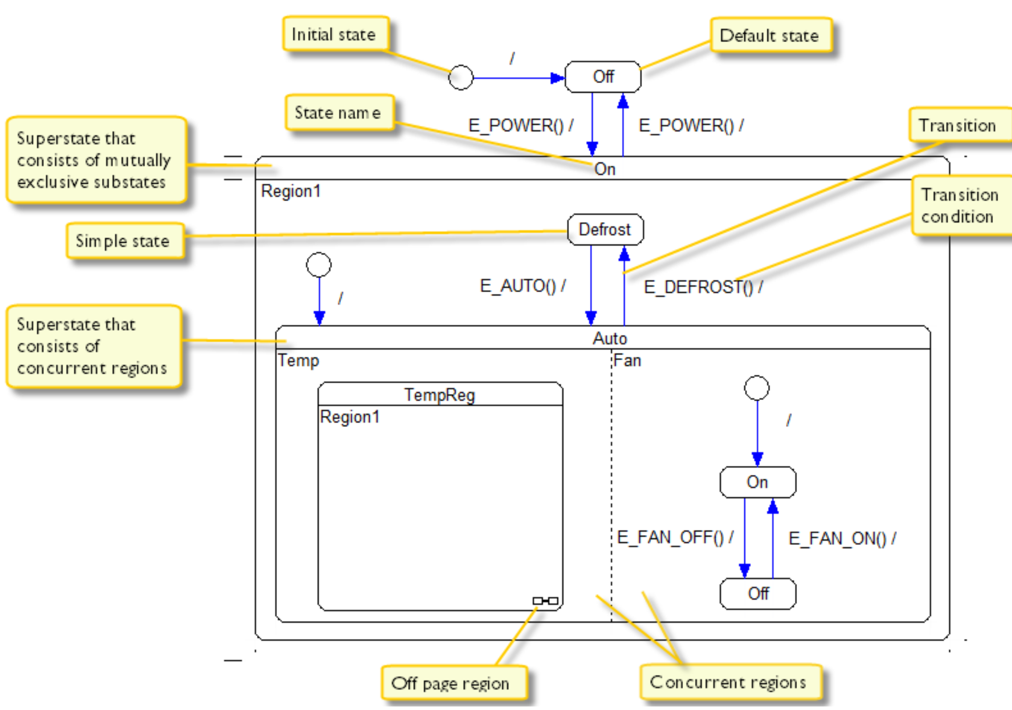
Your state machine design model decides how to react on the input from the environment of your embedded application.

Typically, an embedded system has special-purpose devices such as sensors, actuators, buttons, and displays. The input from the environment can, for example, come via switches and buttons for turning the power on or off, or changing the playback volume. There are sensors for detecting activities, and there is some sort of output, for example to activate, control, and give feedback to the environment. Input from sensors is called *events*, and output is called *actions*. See also the chapters *States* and *Transitions*.

State machine diagrams—the graphical representation

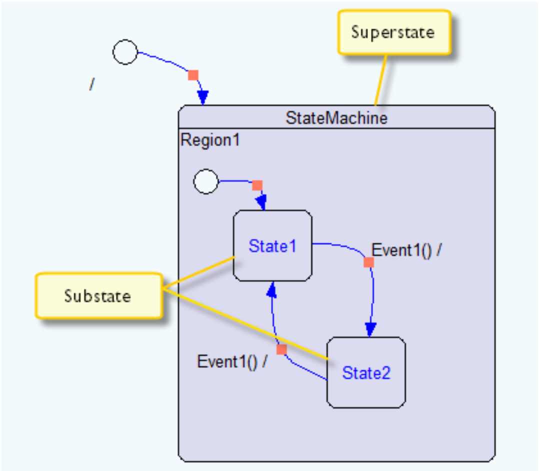
You design your state machine model by drawing a *state machine diagram—the graphical representation of your model—* and naming the objects it contains. The diagrams help you visualize the behavior of your embedded system and provide you with the overview needed for understanding and handling model complexity. Once the diagram has been drawn, you can simulate and verify the state machine to see if it behaves as intended. See also the chapters *Simulation* and *Formal verification*.

This example represents a simple air conditioning unit for a car and illustrates different state types and other components:

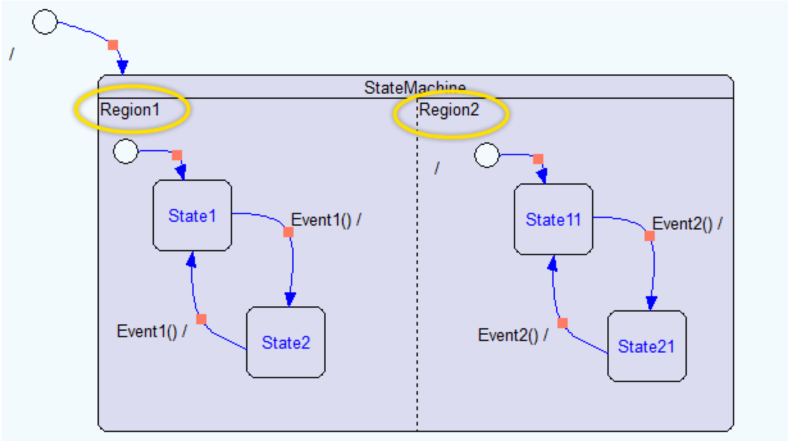


State machine hierarchy and concurrency

IAR Visual State can be used for modeling *hierarchical state machines* as described in the UML standard. Thus, a state machine can contain other state machines. A state that in itself contains one or more state machines is called a *superstate*, and states inside an enclosing superstate is called a *substate*:



A state machine can only be in one state at a given time—the states are *mutually exclusive*. However, if your system must be in more than one state at a time to handle concurrency, you must organize the states in separate *regions*:



In this case, your system can be in more than one state at a given time, for example in `State1` and `State21`.

Regions collect states that belong to a specific state machine, and regions help you modularize your design. They appear in these ways:

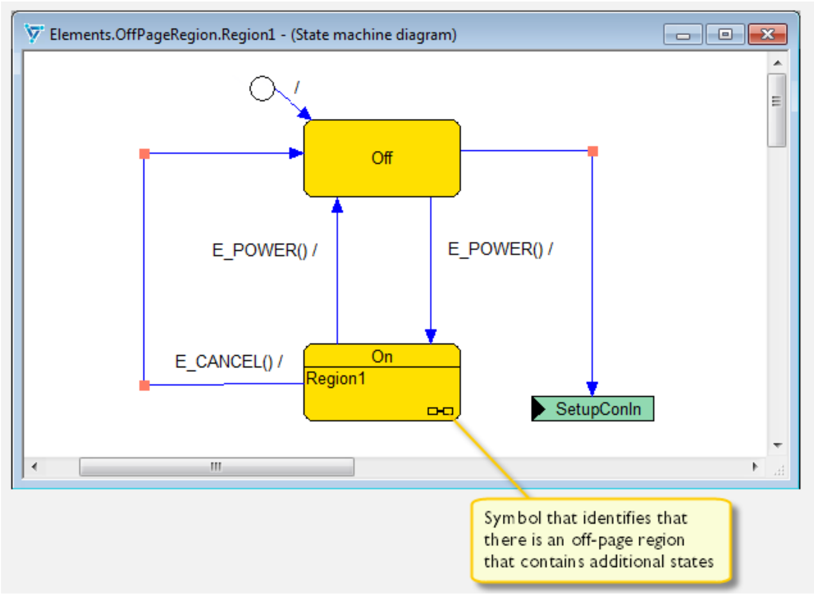
- As concurrent regions in the topstate, in other words, as the top-level state machines in a state machine diagram.
- As one or more regions in *composite states*. See also *Composite state*, page 140.

Organizing complexity using off-page regions

Off-page regions can be used for modularizing complex models. They make it possible to move the advanced control logic of a composite state to another state machine diagram instead of representing it directly in a composite state.

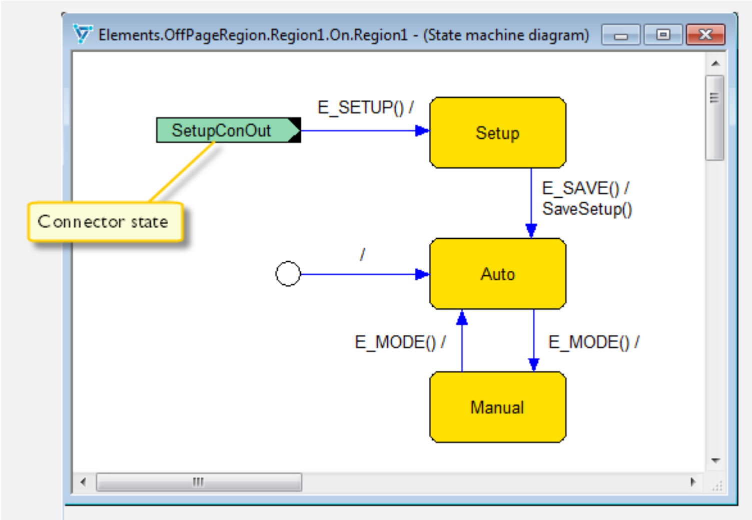
The runtime behavior of a composite state with an off-page region is the same as the runtime behavior of a composite state that contains a state machine directly. The difference between the two constructs is only their graphical representation in the state machine diagram.

This example shows a device that can be in two different states: `Off` and `On`:



The state `On` contains other states (a state machine), which are drawn as an off-page region. The graphical representation of an off-page region in the parent state is a small symbol that indicates that the contained state machine is represented in its own diagram.

This example shows the state machine that is contained in the off-page region in the previous image:



The two images also illustrate that you can create a transition that crosses the diagram boundary from the containing state to the states in the off-page region. This is accomplished by using *connector* states. The transition triggered by the event `E_SETUP` is an example of this. See also *Connector pseudostate*, page 149.

Reuse

IAR Visual State offers several ways of reusing your work, instead of having to design identical or similar elements over and over again. These mechanisms are intended to make reuse easier:

- System instances

  To control multiple identical hardware or software units by means of the same state machine model, you can create multiple system instances. Instead of manually copying the code for each individual unit, you create instances of the data for the unit. See *Reuse of design using system instances*, page 126.

- Stereotypes

  Defining *stereotypes* is a simple way to create states with a uniform look. A stereotype is a named template that captures the size, color, font, entry, and exit reactions of a state. See *Stereotypes for creating states with a uniform look*, page 140 and *Creating states with a uniform look using stereotypes*, page 156.

- Transition element files

  Transition elements can be stored in small, reusable files that contain only transition elements, not states or transitions. You can reuse those transition elements by simply adding the transition element file to a project. See *Transition element files*, page 179.

- State machine templates

  Instead of cutting and pasting to reuse state machines, you can create a *state machine template* of it, and instantiate this template one or more times by creating submachine states. See *Reusing designs using state machine templates*, page 201.

- Copying and pasting state reactions

  Instead of creating identical reactions for states, you can right-click on a state in a Designer diagram and copy the reactions from the state. You can then paste them into another state in the same or a different diagram. See *General Designer windows context menus*, page 298.

RUNTIME BEHAVIOR—MACROSTEPS AND MICROSTEPS

When an event is processed by IAR Visual State at runtime, the following happens in this order:

1 The event is processed and all enabled transitions are executed. In other words, transitions that start in a currently active state, have no guard condition or have guard conditions that evaluate to true, and trigger on the processed event. As a direct consequence, the following happens:

   - Action functions and assignments, as well as entry and exit actions, that are part of the executed transitions are performed.
   - States change to the goal states of the executed transitions.
   - Signals emitted as part of the transition execution are queued up in the signal queue in the order of processing.

2 At this point, there might be *trigger-less* transitions that start in a currently active state and have no guard conditions or whose guard conditions evaluate to true. These transitions are executed in the same way as in step 1. Step 2 can lead to more signals being added to the signal queue.

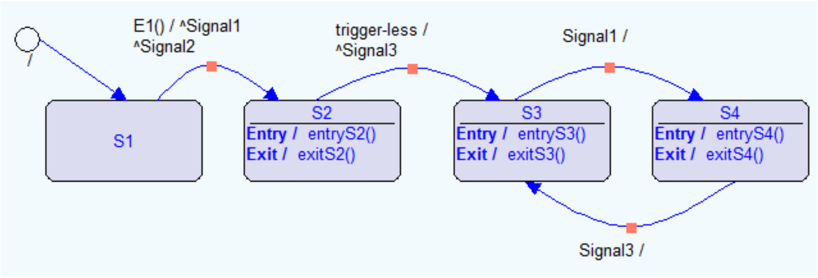
   Step 2 is repeated until no trigger-less transitions are enabled anymore. Note that this might lead to a *livelock*—an unlimited sequence of steps to process trigger-less transitions. The tool does not prevent livelocks, just as a compiler does not prevent infinite loops from being written.

3 If the signal queue is not empty, and if there are enabled transitions that trigger on the first signal in the queue, then this signal is retrieved from the queue and processed in the same way as an external event.

Steps 2 and 3 are repeated until no trigger-less or signal-triggered transitions can be executed anymore. At this point, any signals still in the queue are discarded.

A complete sequence of step 1–3 comprise a *macrostep*; each individual step is called a *microstep*. (Note that it is possible to create models where step 2 and 3 repeat for ever—another source of livelock.)

This diagram exemplifies the runtime behavior:



After sending the `SE_RESET` and `E1` to the system, the model will:

- call these action functions in sequence: `entryS2`, `exitS2`, `entryS3`, `exitS3`, `entryS4`, `exitS4`, and `entryS3`
- discard `signal2`
- be in state `S3` when finished.

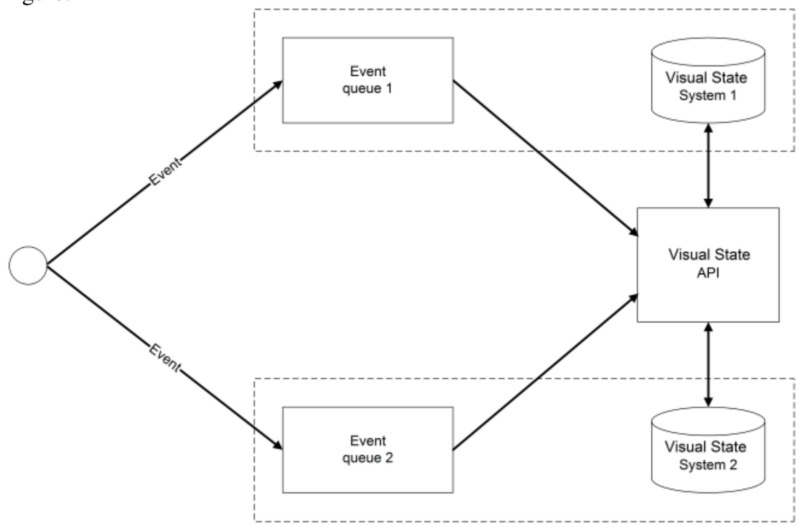
See also *Triggers*, page 169 and *Signal*, page 181.

THE VISUAL STATE SYSTEM

A Visual State *system* is a collection of one or more state machine models. Your Visual State project can contain one or more systems.

On target, the logical unit of a state machine model is the Visual State system. Thus, when an event occurs, it is interpreted on a per system basis. So, although a system might contain more than one hierarchical state machine, the event occurs for all state machines in this particular system.

If the project consists of more than one system and the systems share an event, (or, in other words, they must react to the same event), the event handling mechanism on target must ensure that the event occurs once for each system. This principle is shown in this figure:



The figure shows the interaction between IAR Visual State and the external environment. The event happens physically in the external environment, and is added to the event queues of the individual system. From there, the event is sent into the Visual State API on a per system basis. You must implement these event queues. You can find source code examples for event queue handling provided with your product installation.

System notation

All design elements in IAR Visual State conform to the Unified Modeling Language specification for state machine diagrams. This means that every model you create with IAR Visual State will conform to that specification.

However, when you use *Safe Mode*, a warning will be given during the design process if a non-verifiable element is used. Safe Mode should be used when you want to be sure that the design is verifiable with the Visual State Verificator. See also *Non-verifiable elements*, page 417.

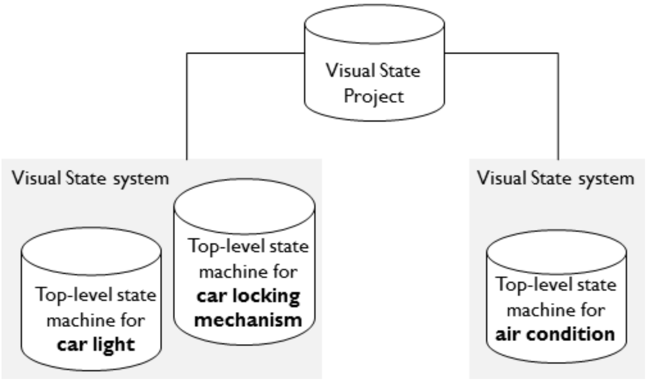
Briefly about organizing your system

IAR Visual State provides many mechanisms for organizing your state machine model. A well organized model not only considers the logic of your application, but also aspects such as concurrent programming, team development, and reuse of designs.

Your Visual State project collects your Visual State systems. For each system you create, you must create a *top-level state machine* for which a file is automatically created (filename extension `vsr`).

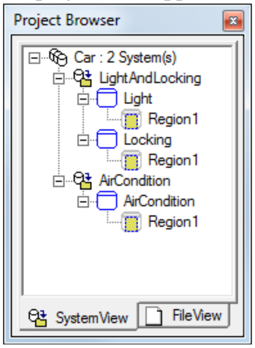
If state machine models are grouped in the same Visual State system, they can be synchronized with each other via *state conditions*, see *State conditions*, page 170 and *Synchronizing one part of the model with other parts of the model*, page 136. Thus, the behavior of one state machine can affect the behavior of the other state machine model(s) within the same system.

For example, if you want to create a state machine for a car light that should react to the behavior of a state machine for the car locking mechanism, it is a good idea to include the two state machines in the same Visual State system. Furthermore, if another state machine—for example a state machine for the air conditioning—is to operate independently of the first two state machines, a separate Visual State system should be used for that state machine model:



In the Designer, you can create a system, for example `LightAndLocking` that contains two top-level state machines, for example `Light` and `Locking`. Also, create a top-level state machine for the air conditioning and insert it in another system, for example `AirCondition`.

In the **Project Browser** window the project contains two systems, `LightAndLocking` and `AirCondition`, respectively. The `LightAndLocking` system contains the two state machines called `Light` and `Locking`. This image shows the state tree structure of the project as it appears in the Designer:



Because the system is split into more than one top-level state machines—thus, saved in separate files—several developers can work in parallel on the same project.

Reuse of design using system instances

If you want to control multiple identical hardware or software units by means of the same state machine model, you can create multiple system instances. Instead of manually copying the code for each individual unit, you create instances of the data for the unit.

At runtime, IAR Visual State is capable of handling more than one instance of each system. Each instance has its own state configuration and its own copy of the internal variables so that the instances are completely independent of each other. Only one instance can be active at a time. The API is used for activating an instance.

Note also that IAR Visual State supports multiple similar variants of your product. For more information, see *Using variants and features*, page 217.

Designing state machines

Before you can start designing your state machine models, you must first create a project with a system and a state machine file, see *Setting up workspaces and projects*, page 75.

This is an example procedure that you can follow when you design state machines:

1. *Identifying and creating events and action functions*, page 127
2. *Identifying and drawing simple states*, page 128
3. *Organizing your states logically*, page 129
4. *Creating transitions between your states*, page 130

5 *Synchronizing one part of the model with other parts of the model*, page 136

Note that the exact order is not mandatory, but when you design it helps to start by looking at the problem, to identify possible actions and states. Most certainly, you will iterate these steps one or more times.

IDENTIFYING AND CREATING EVENTS AND ACTION FUNCTIONS

The requirements of an embedded application determine what your state machine model must react to and how it should react. What input will your system need and what output will it produce? The input is your events and the output is the actions, which you implement as *action functions*. When the events and actions are identified, you have defined the interface of your system and specified the boundaries for what your application should do. See also *Events*, page 179 and *Action function*, page 182.

To create events:

1 In the Designer, open your project.

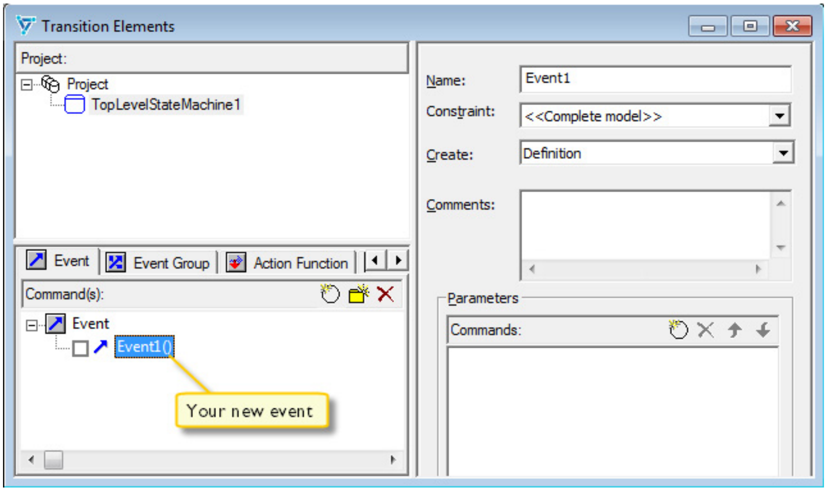
2 Choose **View>Transition Elements** to open the **Transition Elements** window.

3 In the **Project** pane, select your project if you want to create a global element. To create a local element, select the top-level state machine in the tree.

4 In the **Commands** pane, click the **Event** tab.

5 On the **Commands** toolbar, click the **New** button ( ).

A new event with a default name is created in the list:



6 Specify an event name and possibly a description for the event in the **Name** and **Comment** fields, respectively.

Note:

- If you use a descriptive name, you do not need to specify a comment as well.
- At this stage of the design phase you do not have to specify whether it is a definition or a declaration, or specify the parameters. You can specify that later on.

Previously created elements can be dragged from the **Commands** pane to the project or top-level state machine in the **Project** pane. Thus, local elements can become global elements by dragging them to the project in the tree structure. If you press Ctrl while dragging the elements to a another file, the declarations will be added to the destination file, but not the transition element.

You can delete elements by clicking the **Delete** button ( ).

7 Repeat this procedure for the events you need.

To create action functions:

1 Follow the same procedure as for creating events, but click the **Action Function** tab instead, and specify the details for your action function. Note that at this stage of the design phase you do not have to specify the parameters, you can specify them later on.

2 Repeat the procedure for all action functions you need.

IDENTIFYING AND DRAWING SIMPLE STATES

When you have identified your events and actions, it is time to identify the states. Which states will your embedded application have? Identify the states you need and draw a simple state for each one in the state machine diagram. Initially, you do not have to consider how the states logically relate to each other.

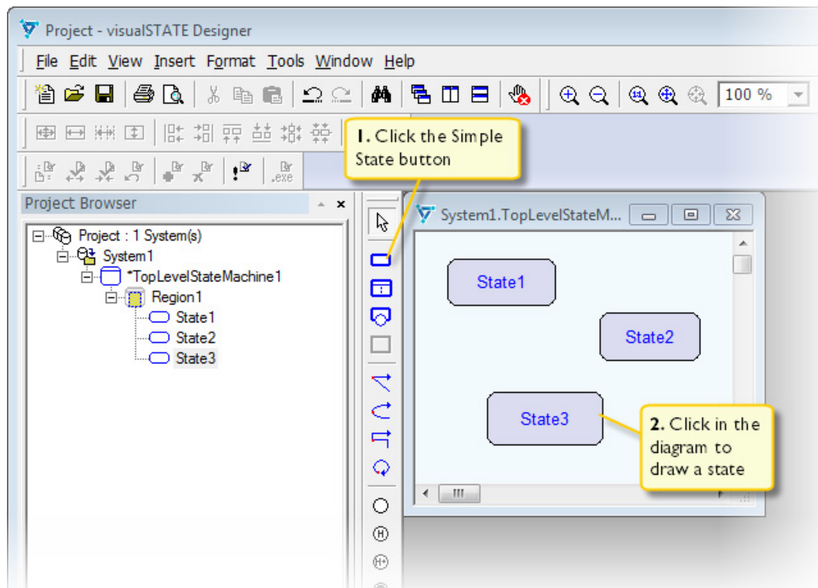
To draw simple states:

1 To open the **State machine diagram** window, double-click the region you want to edit in the **Project Browser** window.

2 On the **Diagram** toolbar, click the **Simple State** button ( ) and click in the **State machine diagram** window. A simple state will be created in the diagram.

To create a state with another size, click in the diagram and hold the left mouse button while you drag a rectangle. Release the mouse button.

3 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.



4 By default a state is given a name `State#`. To change the name, click the default state name, and type a new name that reflects the state of your embedded application.

5 Fill the diagram with the states you need.

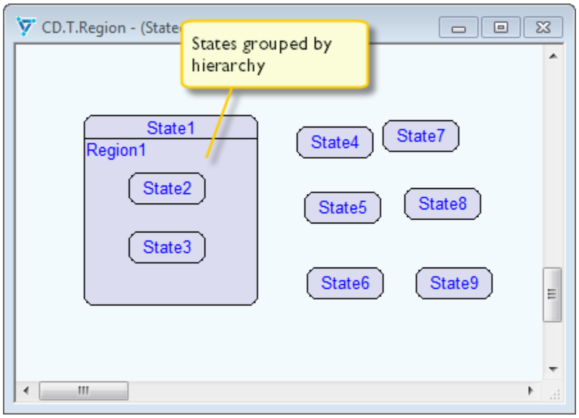
ORGANIZING YOUR STATES LOGICALLY

When you have identified and drawn your states, you must organize them logically; which are related to each other and which are not, and how are they related? You can group your states both by hierarchy and by concurrency, and you will probably do both. Some states will probably be mutually exclusive to other states.

To group your states by hierarchy:

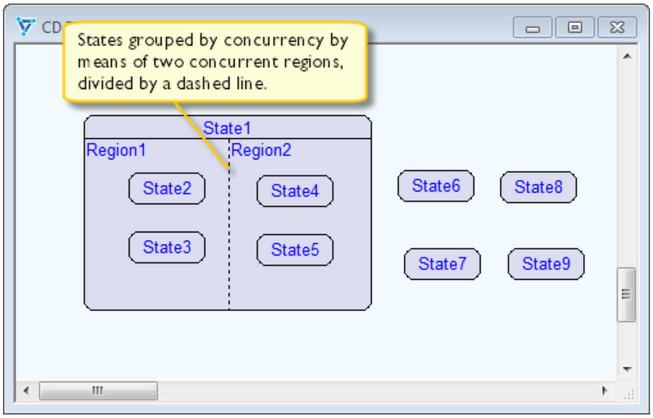
1 Resize the state (for example `State1`) that you consider being a super state to other states (for example `State2`), and simply move `State2` inside `State1`. A region is automatically created in `State1`.

2 Move all states that are related to `State1` by hierarchy to it.



To group your states by concurrency:

1 Assume that `State4` and `State5` are concurrent to `State2` and `State3`. In that case, resize `State1`, right-click `Region1` and choose **Insert Region>Right** from the context menu. `Region2` is created to the right of `Region1`.



2 Move `State4` and `State5` to `Region2`.

CREATING TRANSITIONS BETWEEN YOUR STATES

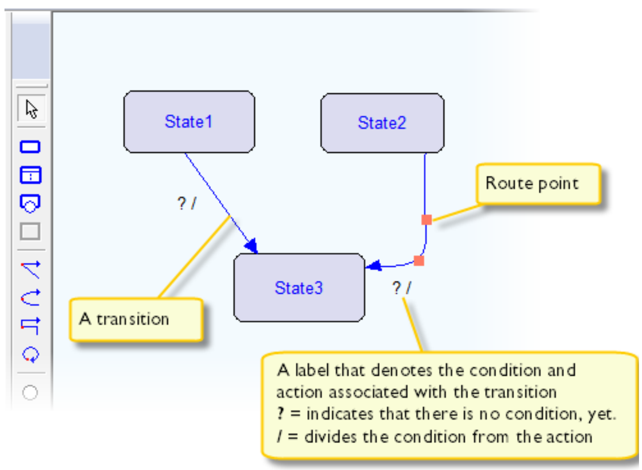
The transitions specify the dynamic behavior of your embedded application. When an event occurs in the state machine environment, the state machine changes its state by performing a *transition*; optionally, an action can be performed. You draw the transition line from the source state to the destination state and associate a condition and action with the transition.

To control the route of the transition, you might want to add route points to your transitions, either manually or by letting the Designer automatically determine the route of the transition. The route point is a handle which changes the route of the transition when you drag it, which can be useful for complex diagrams.

See also *Introduction to transitions*, page 167.

To draw the transition:

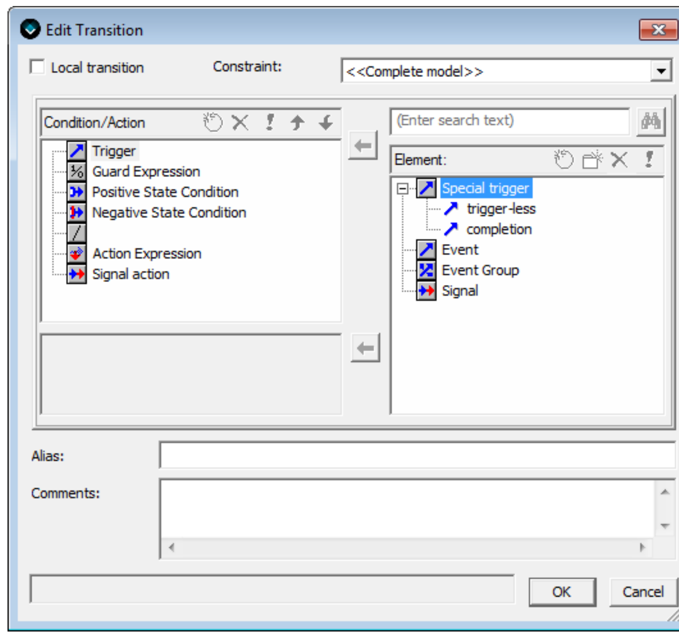
1. On the **Diagram** toolbar, click the **Transition** button ().
2. In the state machine diagram, click the source state. The frame of the source state is highlighted and a hook point is displayed.
3. To start drawing the transition, move the cursor to the frame of the destination state; the frame is highlighted. Click the destination frame. The transition line is drawn between the states.



4. Deactivate the transition tool by right-clicking in the diagram.

To specify the condition and action:

1 In the state machine diagram, select the transition for which you want to specify the condition and action, right-click and choose **Edit Transition** from the context menu (or double-click the transition in the diagram).



Here you can specify the condition and the action, add comments, and specify an alias for your transition.

2 Select an item in the **Condition/Action** pane:

- Use the first four items to specify the condition: **Trigger**, **Guard Expression**, **Positive State Conditions**, and **Negative State Conditions**.
- Use the last two items to specify the action: **Action Expression** and **Signal Action**.

When you have selected an item in the **Condition/Action** pane, the valid *transition elements* for that item will appear in the **Element** pane.

For example, if you select **Trigger**, the elements **Special trigger**, **Event**, **Event Group**, and **Signal** appear in the **Element** pane.

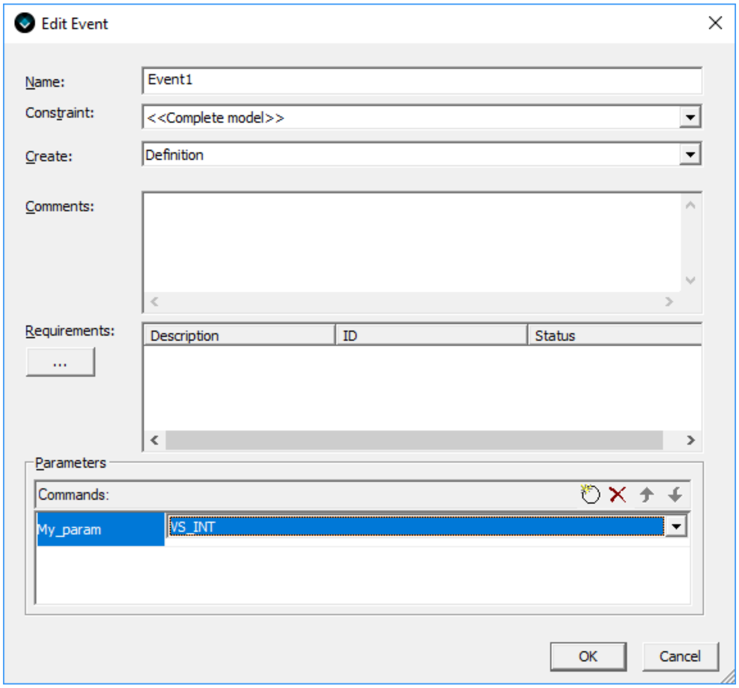
For information about available transition elements per item, see *Introduction to transitions*, page 167. See also *Transition elements*, page 177.

3 In the **Element** pane, select what you want to apply to your condition or action. For example, an **Event**.

If you have created events previously, they will be listed automatically in the **Elements** pane. Otherwise, you should create the event (or any other transition element) now.

4 To create new transition elements, click the **New** button ( ) on the **Element** toolbar. An edit dialog box is displayed. The dialog box looks slightly different for the various transition elements, but the procedure is the same.

5 To create a new event, click the **New** button ( ) on the **Command** toolbar. For events, the dialog box looks like this:

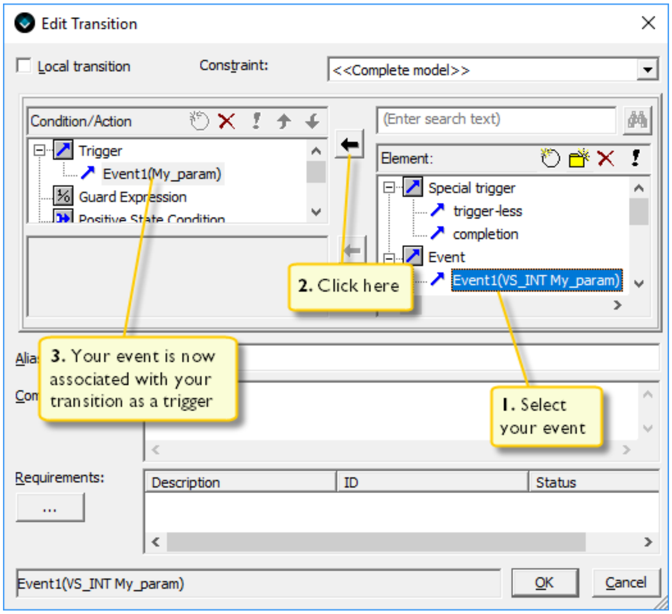


Specify your event details and click **OK**.

Note that you can also use the **Transition Elements** window to create and edit your transition elements. See *Transition Elements window*, page 295

6 In the **Edit Transition** dialog box, your newly created transition element—the event `Event1` with the parameter `My_param`—now appears in the **Elements** pane.

7 Select your transition element and click the black left arrow (or double-click the element) to associate your transition element with your condition or action.

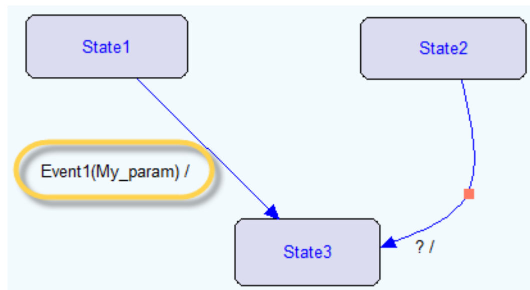


Your element appears in the **Condition/Action** pane.

8 For each item in the **Condition/Action** pane that you want to be part of your condition or action, repeat steps 1–7.

You can add as many transition elements to your conditions and actions as you want, and change their order by clicking the Up and Down Arrow buttons on the toolbar. To delete an item from the condition or action, select the item and click the **Delete** button (✕).

9 When you have finished creating your condition or action, click **OK**. The element appears next to your transition arrow in the state machine diagram.



To edit the condition and action, you can right-click the transition and choose **Edit Transition** from the context menu.

To edit the transition elements, choose **View>Transition Elements**. See *Creating a transition element*, page 184.

To insert route points manually:

1 To insert a route point manually while you draw a transition, click in the diagram outside the target state. This will insert a route point for the transition and let you continue to draw the transition.

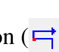
To remove the most recently inserted route point, right-click in the diagram.

2 After completing a transition, you can clone a route point by pressing Ctrl while you click and drag the route point.

To delete a route point, drag it and drop it on another one.

To let the Designer automatically determine the route of the transition:

1 Choose **Tools>Settings>Transition** and make sure **Auto format orthogonal transitions** is selected.

2 On the **Diagram** toolbar, click the **Orthogonal Transition** button ().

3 In your diagram, click the source state and then the destination state; the transition will be drawn automatically in such a way that any states in between will be avoided.

4 To help the Designer draw the desired route of the transition automatically, click at specific points in the diagram to guide the Designer.

SYNCHRONIZING ONE PART OF THE MODEL WITH OTHER PARTS OF THE MODEL

Typically you want to synchronize state machines that depend on each other, for example a car light that depends on the car locking mechanism.

There are various mechanisms that you can use for synchronizing your state machines:

- Signals

  A signal triggers a transition. For example, when a car locking state machine decides that the car is locked, a signal action can be sent as a synchronization with the car light state machine. See *Signal*, page 181.

- State conditions

  State conditions ensure that another state machine within the same Visual State system satisfies when another state machine is in a specific state, not in that state, or in a combination of states. This means that you can synchronize one state machine with another state machine; for example, the car light will not change state until the car lock state machine is in the state locked. See also *State conditions*, page 170.

- Trigger-less transitions

  Trigger-less transitions are special transitions that do not have an explicit trigger, but that usually have a guard condition that can depend on other state machines. See also *Trigger-less transitions*, page 173.

To create a signal, state condition, or trigger-less transition, see *Creating a transition element*, page 184.

States

- Introduction to states
- Working with states
- Working with composite states and regions

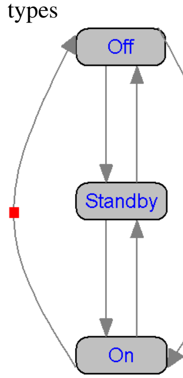
Introduction to states

Learn more about:

BRIEFLY ABOUT STATES

A state represents the current situation in the system. A state in a state machine is an abstract mapping of one or more states. For example, a printer can be `Off`, in `Standby`, or `On`.

In the state machine diagram, states are drawn with a different symbols for different state types



A state machine does not have to map all the possible physical states of the underlying hardware, only the states that are important to the model.

A state machine moves from one state to another state if a specific *event* occurs—for example, pressing a button—by performing a *transition*. See also *Transitions*, page 167.

In some situations you might want something to happen but without changing states. In this case you can create *internal reactions*. Internal reactions behave like transitions but they do not change states. For more information, see *State reactions*, page 150.

Note: In the UML standard, states are referred to as vertexes.

Overview of available states

There are different types of states, with their own graphical representations and logical meanings:

- Simple state—a state that does not contain any other states or regions. See *Simple state*, page 140.
- Composite state—a state that consists of one or more regions, which contain other states. A composite state is used for representing the top-level state machine. See *Composite state*, page 140.
- Initial state—represents a default state that is the source for a single transition to the *default state* of a composite state. See *Initial state*, page 141.
- Shallow history state—a shorthand notation that represents the most recent active substate of its containing state (but not the substates of that substate). See *Shallow history pseudostate*, page 143.
- Deep history state—a shorthand notation that represents the most recent active configuration of the composite state that directly contains this pseudostate; that is,

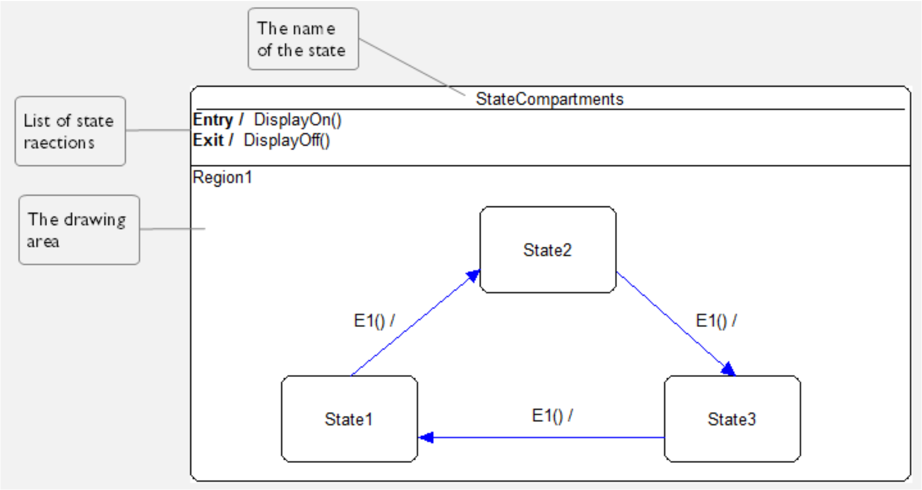
the state configuration that was active when the composite state was last exited. See *Deep history pseudostate*, page 147.

- *Join and fork states*—join states serve to merge several transitions coming from source states in different concurrent regions, whereas fork states serve to split an incoming transition into two or more transitions terminating on destination states in different concurrent regions. See *Join and fork pseudostates*, page 148.
- *Junction states*—states that are used to chain together multiple transitions. See *Junction pseudostate*, page 149.
- *Connector states*—states used for constructing compound transitions that cross an off-page boundary. See *Connector pseudostate*, page 149.
- *Choice states*—states used for setting up dynamic choice between a number of transition paths, where the path to take depends on what the actual values are before continuing from the choice state. See *Choice state*, page 150.

State compartments

Graphically, a state can consist of up to three different compartments, which specify (from top to bottom):

- The name of the state.
- A list of the state reactions of the state, see *State reactions*, page 150.
- The area where you draw any substates.



The two latter compartments are optional. Whether they are present or not depends on how the state was created.

Stereotypes for creating states with a uniform look

Defining *stereotypes* is a simple way to create states with a uniform look. A stereotype is a named template that captures the size, color, font, entry, and exit reactions of a state. Hierarchical information is not captured. A project can use as many stereotypes as necessary.

The typical use for stereotypes is when you want states to have some behavior or property in common.

See also *Creating states with a uniform look using stereotypes*, page 156.

SIMPLE STATE

A simple state is a state at the lowest level in the state hierarchy, and it does not contain any other states or regions. Graphically, a simple state is represented like this:

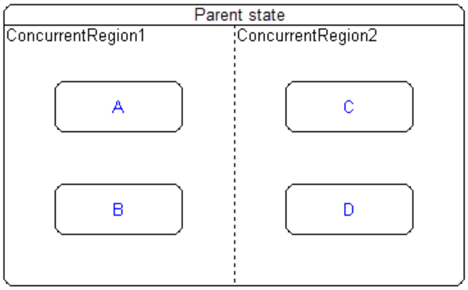
A simple state can have state reactions, see *State reactions*, page 150.

COMPOSITE STATE

A composite state is a state that has at least one *region*. A composite state can consist of:

- One or more concurrent regions
- Mutually exclusive states.

This figure illustrates a state with concurrent regions, where the states inside each concurrent region are mutually exclusive:



Each of the concurrent regions represents a state machine, and the individual concurrent region is active as long as the containing composite state is active. The concurrent regions are separated by dashed lines. For more information, see *State machine hierarchy and concurrency*, page 119.

When a composite state becomes active, one (and only one) of the states in each region in it, becomes active. A composite state can be entered either implicitly or explicitly depending on the transition that causes the state to become active.

One example of a composite state is the state that represents the top-level state machine. Such states will always be active. Consequently, the only allowed state reactions in such a state are entry reactions, which will be fired upon reset, and internal reactions. Exit reactions are not allowed, because they will never be executed. The top-level state machine is saved in its own file and can thus serve as a building block for developer teams. See also *Briefly about organizing your system*, page 124.

A transition can cause a region in a composite state to be entered in one of four ways:

1. *Default entry*: If a transition does not end directly in a region but on the composite state of the region, the region enters the default state, indicated by the transition leaving the initial state in that region.
2. *Explicit entry*: If a transition ends directly on a simple state (or a composite state) in a region, the region enters that state, and in the case of a composite state, these four ways are applied recursively.
3. *Shallow history entry*: If the transition terminates on a shallow history state, the active substate becomes the most recently active substate prior to this entry, unless the most recently active substate is the final state or if this is the first entry into this state. In the latter two cases, the default history state is entered. This is the substate that is destination of the transition originating from the history pseudostate. See *Shallow history pseudostate*, page 143.
4. *Deep history entry*: The same rule as for shallow history entry except that the rule is applied recursively to all levels in the active state configuration below this one. See *Deep history pseudostate*, page 147.

INITIAL STATE

An initial state represents a default state that is the source for a single transition to the default state of a composite state. There must be exactly one such initial state in every region of a composite state.

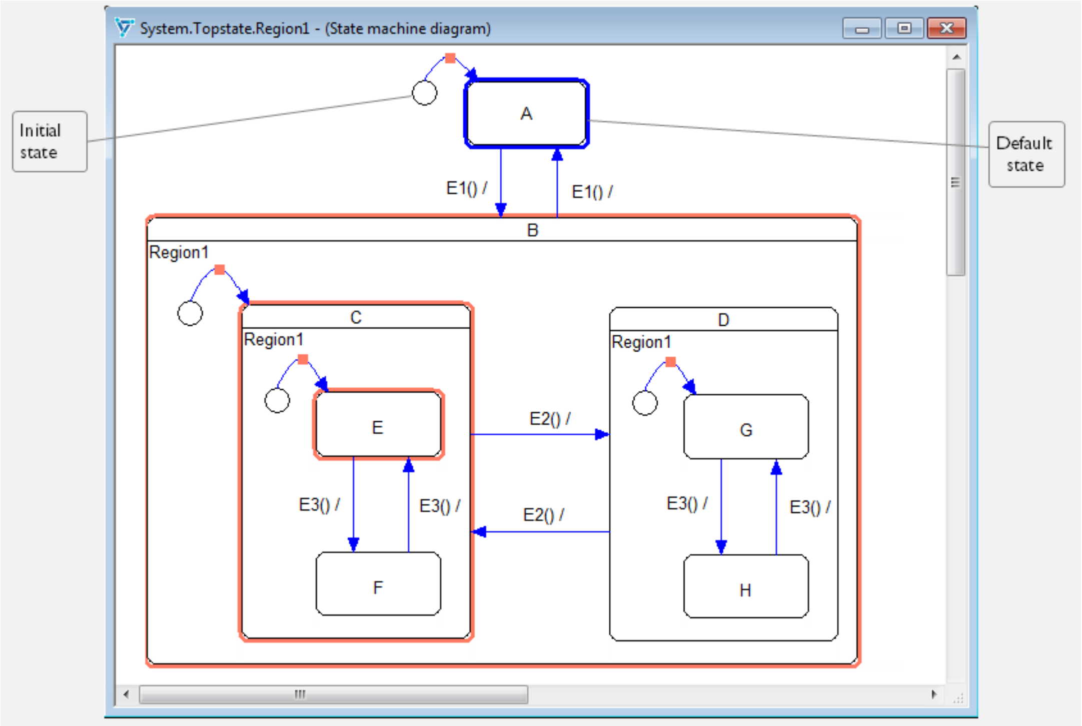
The transition that leaves the initial state in a region is fired, and the default state—the state the transition from the initial state points to—is entered when:

- The owning state machine becomes active, and

- The transition that causes the composite state that owns the region does not have an explicit destination state in this region or to one of its descendants.

For example, if the parent state machine is a composite state, and the transition causing the machine to become active has this composite state as its destination state, the default state in every region becomes active.

This is an example of an initial state and a default state;



In this figure, state `A` is the default state, which is indicated by the transition from the initial state into state `A`. This means that upon reset, state `A` is entered.

When event `E1` occurs, the state machine will go from state `A` to composite state `B`. Because the state machine contained in composite state `B` is activated, it will in `Region1` enter `C` by taking the transition from the initial state in `Region1`, and in `Region2` it will enter its default state `C`. State `C` represents a state machine which will enter state `E`.

SHALLOW HISTORY PSEUDOSTATE

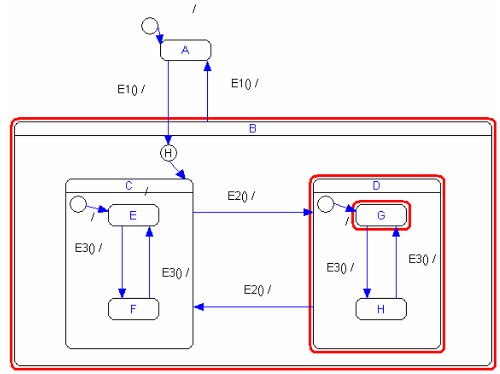
A *shallow history pseudostate* is a shorthand notation for the most recent active substate of its containing region (but not the substates of that substate). A composite state can have any number of shallow history states.

A transition coming into one of the shallow history states in each region is equivalent to a transition coming into the most recent active substate of a state. By having more transitions going into different shallow history states, you can let different conditions decide which default state you enter for each shallow history state. Exactly one transition must originate from each shallow history state to the default shallow history state. This transition is taken in case the composite state has never been active before for this region or the history has been cleared when this shallow history state is entered.

Shallow history states for this region behave as described in the UML specification. For a shallow history state to have any effect, the transition must go to the history state.

To use a shallow history state, place one at the level where you want it and make a transition point to it. There should be exactly one outgoing transition from the shallow history state to another state. The first time the transition to the shallow history state is taken, the state pointed to by the outgoing transition from the shallow history state is entered. The next and every following time the transition is taken the state machine remembers its previous state in this region, until a final state in this region inside the state machine is entered and the history is cleared for this region.

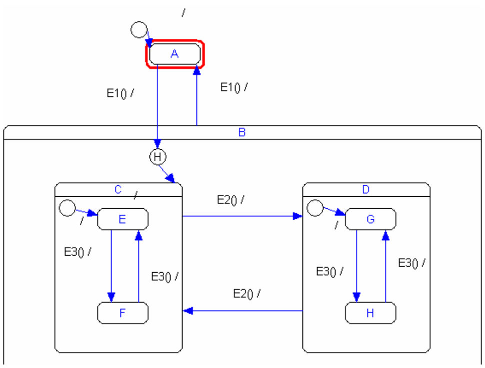
This example shows the same example as the one shown in *Initial state*, page 141. The difference between the two figures is that in this figure, state `B` contains a shallow history state and not an initial state. The first time state `B` is entered, the result is the same as for the previous figure.



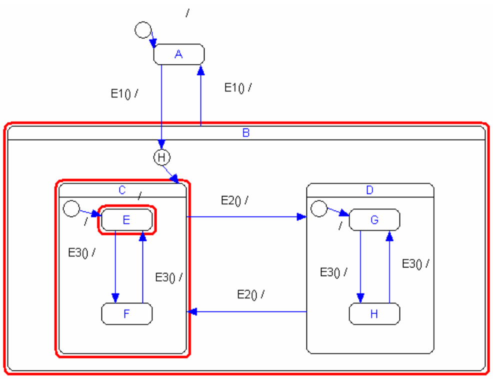
When event `E1` occurs, the state machine goes from state `A` to composite state `B`. Because the state machine contained in composite state `B` is activated, in `Region1` it enters `C` by taking the transition from the shallow history state in `Region1`, and in `Region2` it enters its default state `C`. State `C` represents a state machine which will enter state `E`.

The series of screen captures below shows the results of sending the following sequence of events upon reset: `E1()`, `E2()`, `E3()`, `E1()`, `E1()`;

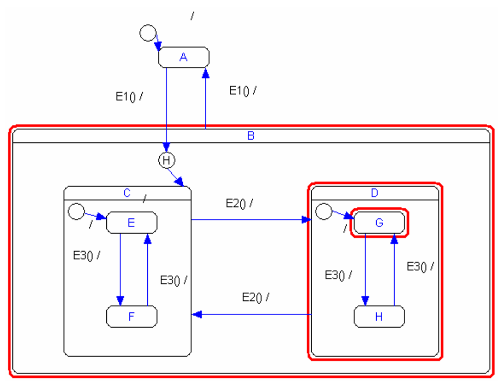
1 Reset of state machine => The state machine will be in the default state `A`.



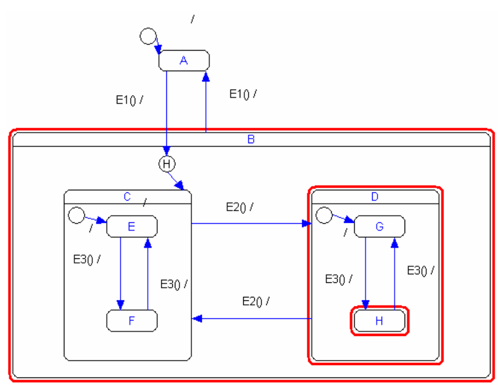
2 Event `E1` is sent => The state machine enters state `B` and its default state `C`. State `C` represents a state machine which will enter state `E`.



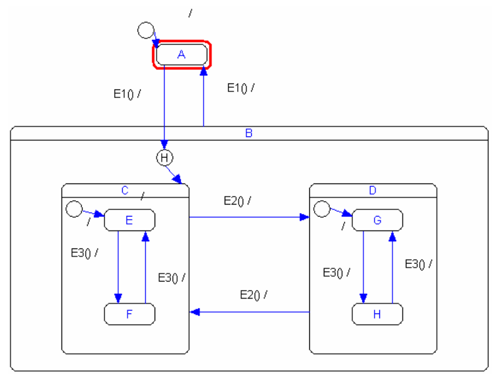
3 Event E2 is sent => The state machine enters state G which is the default state of the state machine that is represented by state machine D.



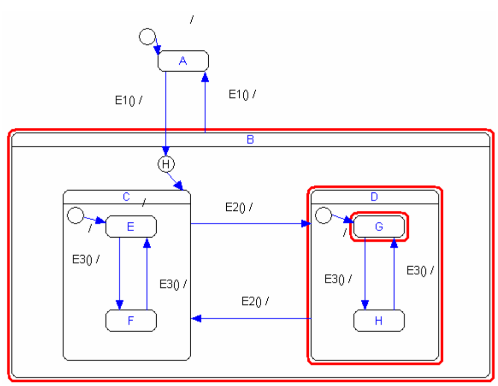
4 Event E3 is sent => The state machine enters state H.



5 Event E1 is sent => The state machine enters state A.



6 Event E1 is sent => The state machine enters state B. Because state B contains a shallow history state, it will enter state D.



The difference between using an initial state and a shallow history state can be seen from the fact that when the state machine reenters state B, state D is entered and not state C. Thus, state B has a history of the state it was in when it was left.

You can have both a shallow history state and an initial state in a state. All transitions that go to the border of the state enter the initial state, and all transitions that go to the shallow history state will go to the history state, but will lose any substates.

DEEP HISTORY PSEUDOSTATE

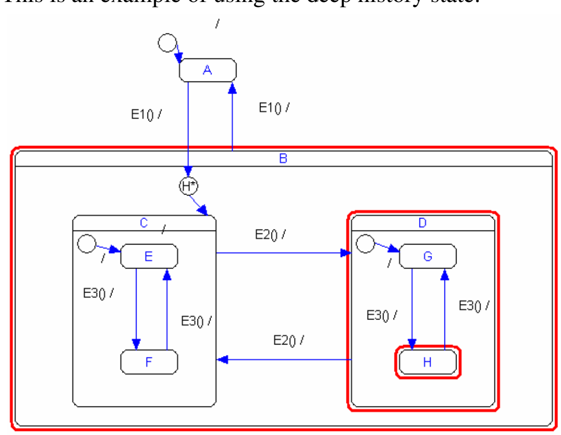
A *deep history pseudostate* is a shorthand notation for the most recent active configuration of the composite state that directly contains this pseudostate; that is, the state configuration that was active when the composite state was last exited.

Deep history states behave as described in the UML specification. For a deep history state to have any effect, the transition must go to the history state.

To use a deep history state, place one at the level where you want it and make a transition point to it. There should be exactly one outgoing transition from the deep history state to another state. The first time the transition to the deep history state is taken, the state pointed to by the outgoing transition from the deep history state is entered. The next and every following time the transition is taken, the state machine remembers its previous state in that region, until a final state in that region inside the state machine is entered and the history is cleared for that region.

The behavior obtained by using the deep history state with a state machine is the same as the behavior obtained by using the shallow history state with the same state machine and all state machines below in the hierarchy.

This is an example of using the deep history state:



The example is the same as the one in *Shallow history pseudostate*, page 143, except for that state `B` has been changed to contain a deep history state.

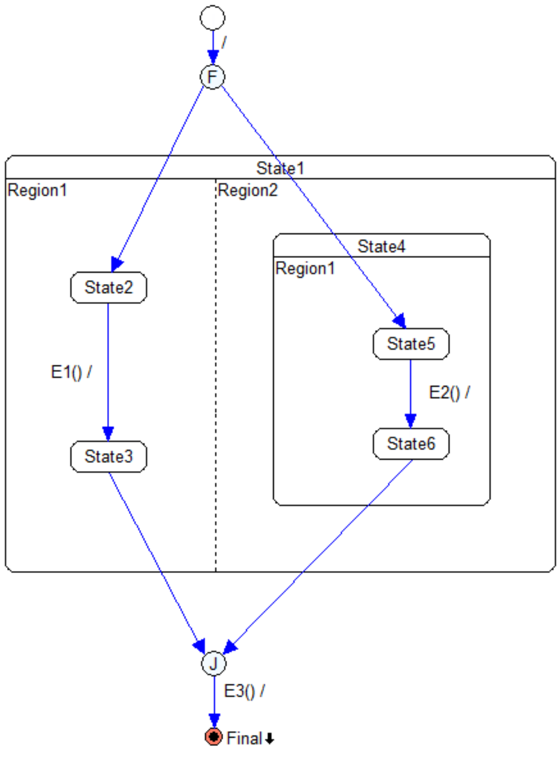
Assume that the following sequence of events is sent: `E1()`, `E2()`, `E3()`, `E1()`, `E1()`. The effect will be that upon re-entering state B, the state machine “remembers” the last state it was in at all lower levels in the hierarchy before it was left. In contrast to the shallow history example, your model now “remembers” that the state machine represented by state `D` was in state `H`.

Note: If a deep history state is used, an initial state must still be applied to each state machine in the hierarchy below the deep history state.

JOIN AND FORK PSEUDOSTATES

Join pseudostates merge several transitions coming from source states in different concurrent regions. The transitions entering a join state cannot have conditions and actions.

Fork pseudostates split an incoming transition into two or more transitions terminating on destination states in different concurrent regions. The transitions that go out from a fork state cannot have conditions or actions.

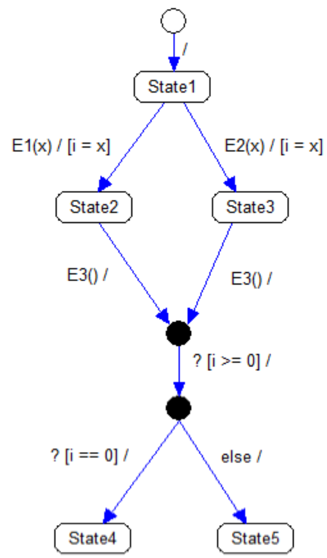


Join and fork states have the same behavior as described in the UML specification.

See also *Drawing fork and join states*, page 158.

JUNCTION PSEUDOSTATE

Junction states combine incoming and outgoing transitions, and construct compound transition paths between states. For example, a junction can converge multiple incoming transitions into a single outgoing transition representing a shared transition path (this is known as a merge). Conversely, they can split an incoming transition into multiple outgoing transition segments with different guard conditions. This realizes a static conditional branch, which means that the evaluation of the guard conditions do not depend on any action expressions located on transition segments before the junction.



Junction states behaves as described in the UML specification.

Note: One of the outgoing transitions on a junction state can be an *else* transition. This means that for each of the incoming transitions, this outgoing transition will be taken when the combined condition sides of the other outgoing transitions are not fulfilled. See *Else transitions*, page 174. Se also *Drawing a junction state*, page 159.

CONNECTOR PSEUDOSTATE

Connector states are similar to junction states. Connector states exist in (connected) pairs with identical names. Connector states create compound transitions that cross an off-page boundary. See also *Drawing a connector state*, page 157.

CHOICE STATE

Choice states are useful when you want to have a dynamic choice between a number of transition paths, where the path to take depends on what the actual values are before continuing from the choice state.

For example, if you have assigned the value of the temperature to an internal variable on the ingoing transition to a choice state, you can have guards on the outgoing transitions that say, for example `[temp < 10], [temp >= 10 && temp < 20] [else]`, which means that the path out depends on the temperature you stored when entering the choice state. Reaching the *choice state* results in a *dynamic* evaluation of the guards on the outgoing transitions. Which outgoing transition to take from a choice state depends on the result from taking the ingoing transition to the choice state.

Exactly one of the outgoing transitions must evaluate to true, otherwise there is a contradiction. You should make one of the outgoing transitions on a choice state as an *else* transition. This means that this outgoing transition will be taken when none of the other outgoing transitions are fulfilled. See *Else transitions*, page 174.

It is good to mark the transitions going out from a choice state as *trigger-less*. The outgoing transition cannot have a trigger, so by marking it as trigger-less you make it clear in the design that you have considered that. See *Trigger-less transitions*, page 173.

See also *Drawing a choice state*, page 159.

STATE REACTIONS

State reaction is a common term used for *internal reaction* (transition), *entry reaction*, and *exit reaction*. State reactions have in common that they are not drawn as lines in the diagram, but appear as conditions and actions inside the state, divided by a `/` (slash) character.

State reactions are fired or can be fired in the following cases:

- When the state machine is entered.
- While the state machine is in a state.
- When the state machine is exited.

Like a transition, a state reaction consists of a condition and an action, but in contrast to transitions, there are certain transition elements that are not allowed in state reactions.

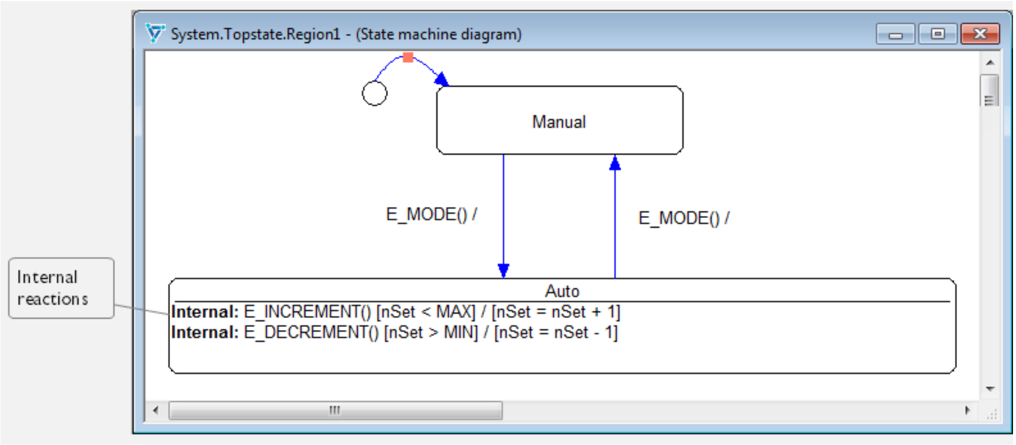
You create state reactions by using the **Edit State** dialog box. See *Creating a state with a state reaction*, page 153.

Internal reaction

An internal reaction is a transition that fires without leaving the state and therefore (in contrast to a transition) does not cause any exit or entry reactions itself. Thus, it has no source or destination state.

Note: Drawing a transition from a state and back to the same state (a self-transition) does not have the same logical implication as an internal reaction. An internal reaction never leaves the state and will thus not cause any exit and entry reactions to be executed. In contrast, a self-transition will actually leave and re-enter the state.

This example illustrates how two internal reactions adjust the value of a variable:



Entry reaction

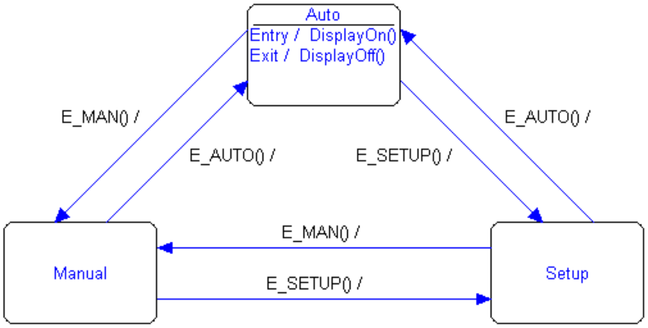
An entry reaction is an action that will be executed each time a state is entered.

Entry reactions offer these advantages:

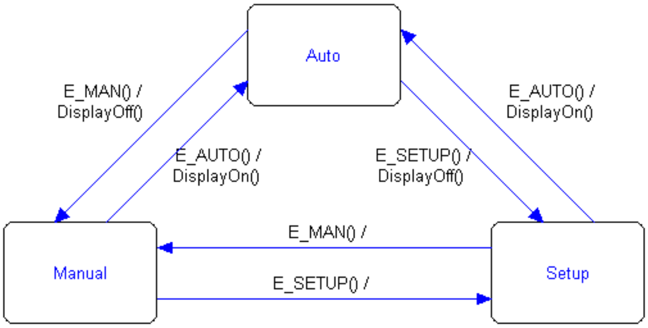
- It is easier to place the actions in the state instead of having to place them on each individual transition that enters the state. This also reduces the risk of errors.
- It gives a much simpler graphical model because the actions are represented only once.
- When a model is modified, the action is automatically associated with a new entering transition.

Graphically, an entry reaction is marked with the keyword `Entry`. No conditions are allowed on entry reactions. Entry reactions are used instead of placing the actions on each transition that enters the state. Thus, initialization should be designed as entry reactions.

The following two state machine diagrams implement exactly the same behavior. However, in the first diagram, entry and exit reactions are placed in the state `Auto`:



In the following diagram, the entry and exit reactions are placed on the individual transitions, giving a more complex graphical model:



Exit reaction

An exit reaction is an action that executes each time a state is exited.

Exit reactions offer these advantages:

- It is easier to place the actions in the state instead of having to place them on each single transition that exits the state. This also reduces the risk of errors.
- It gives a much simpler graphical model because the actions are represented only once.
- When a model is modified, the action is automatically associated with a new exiting transition.

Graphically, an exit reaction is marked with the keyword `Exit`. No conditions are allowed on exit reactions. Exit reactions are used instead of placing the actions on each transition that exits the state.

Working with states

What do you want to do?

- *Creating a state with a state reaction*, page 153
- *Creating states with a uniform look using stereotypes*, page 156
- *Drawing a connector state*, page 157
- *Drawing initial, shallow history, and deep history states (pseudostates)*, page 157
- *Drawing fork and join states*, page 158
- *Drawing a junction state*, page 159
- *Drawing a choice state*, page 159

See also:

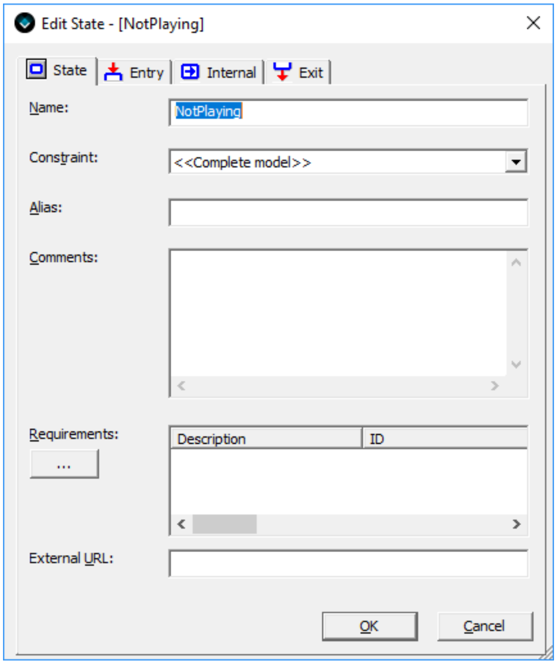
- *Identifying and drawing simple states*, page 128
- *Working with composite states and regions*, page 159
- *Drawing an entry (exit) point state*, page 210

CREATING A STATE WITH A STATE REACTION

When you have drawn a state, you can specify its behavior as a state reaction—internal reactions, or actions at entry and exit.

1. In the Designer, open your project.

2 In the state machine diagram, double-click the state for which you want to specify a state reaction. The **Edit State** dialog box is displayed. Alternatively, right-click the state and choose **Edit State** from the context menu.



Here you can change state name, specify an alias name, and specify entry, exit, and internal reactions. You can also create the transition elements that you need for your state reaction.

3 To create the reaction, click the appropriate tab in the **Reaction** pane, for example **Internal**.

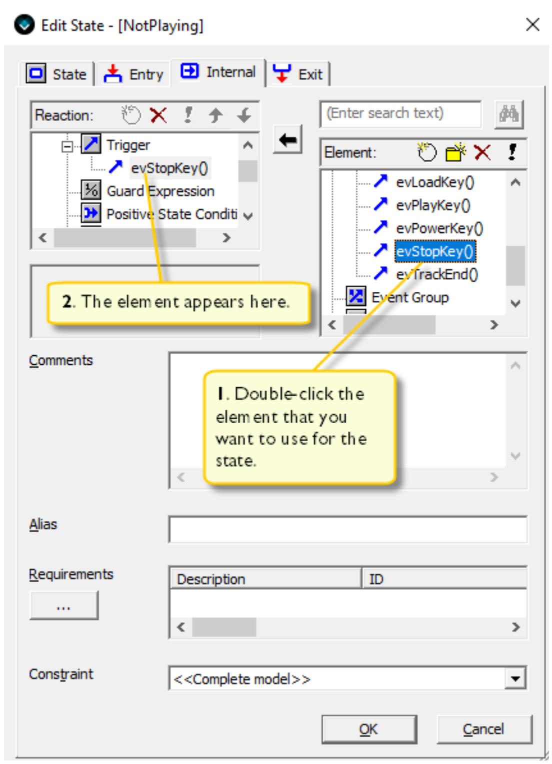
4 To create a reaction, click the **New** button () on the toolbar. A list of possible reaction elements appears in the **Reaction** pane.

5 In the **Reaction** pane, click the type of state reaction you want to use. For example, click **Trigger** in the list, which means that the Trigger element type appears in the **Element** pane.

However, before you can select a specific trigger, you must populate the list by defining your triggers. You can do this in two ways:

- Click the **New** button in the **Element** pane header and specify the properties in the dialog box that appears. When you click **OK**, the defined element appears in the list of elements.
- Use the **Transition Elements** window, see *Creating a transition element*, page 184. Use this window also if you want to edit your elements.

6 When you have populated the list with your triggers, double-click the trigger you want to use. The element will be moved to the **Reaction** pane and applied to the state reaction.



You can add as many elements as you want to, change the order in the reaction list by clicking the Up and Down arrows, and delete elements by clicking the **Delete** button on the toolbar.

See also *Specifying arguments for action function parameters*, page 185.

CREATING STATES WITH A UNIFORM LOOK USING STEREOTYPES

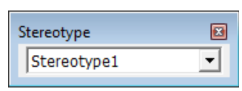
Defined stereotypes can be found on the **Stereotype** toolbar, see *Stereotypes for creating states with a uniform look*, page 140.

To add a new stereotype:

1 Right-click on an existing state whose properties and behavior your want to base other states on and choose **New Stereotype** from the context menu.

2 Type a name for the new stereotype in the dialog box that is displayed and click **OK**.

3 The new stereotype is now the active stereotype in the **Stereotype** toolbar.



When the stereotype toolbar has an active stereotype, any new state that you create will inherit properties and behavior from the active stereotype.

4 To activate another stereotype, choose it from the **Stereotype** toolbar dropdown menu. To revert to creating new states that are not based on a stereotype, activate the **<<none>>** stereotype.

Stereotypes are saved in a separate file in the same folder as the project, and with the same file name as the project file, but with the filename extension `stereotypes`. If a stereotype file is found when a project is loaded, the stereotypes and the active stereotype are loaded from this stereotype file.

To modify a stereotype:

1 Right-click on an existing state whose properties and behavior you want to use instead, and choose **New Stereotype** from the context menu.

2 In the dialog box that is displayed, type the name of the stereotype you want to replace and click OK.

Note: A state that is created from a stereotype copies the information from the stereotype and loses the connection to the stereotype. If the stereotype is modified, states already created using the stereotype do not change.

To delete an existing stereotype, you must manually edit the `*.stereotypes` file.

DRAWING A CONNECTOR STATE

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. Connector states are useful when you must draw a transition between points that are far from each other in the same diagram, or when you must draw a transition to/from the contents of an off-page region. See also *Connector pseudostate*, page 149.

1 In the Designer, open your project.

2 On the **Diagram** toolbar, click the **Connector State Pair** button ( ), and click in your state machine diagram where you want to insert the connector states. Insert two connector states.

3 Draw a transition from the connector states to ordinary states:

4 The states in a connector pair must have the same name to be connected. To connect the selected state with another state, right-click and choose **Select Buddy** from the context menu.

To rename a connector state, click the state and type a new name. Press Enter to finish. When you rename one connector state, the buddy is renamed too, if it has a buddy.

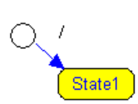
To find the other connector state in a pair, click the connector state and choose **Go to Buddy** from the context menu.

DRAWING INITIAL, SHALLOW HISTORY, AND DEEP HISTORY STATES (PSEUDOSTATES)

1 In the Designer, open your project.

2 On the **Diagram** toolbar, click the **Initial State** ( ), **Shallow History** ( ), or **Deep History State** button ( ). In the diagram, click where you want to insert the pseudostate.

3 Draw a transition from the inserted pseudostate to the state that is to be the default state, in the same way that you draw transitions between states.

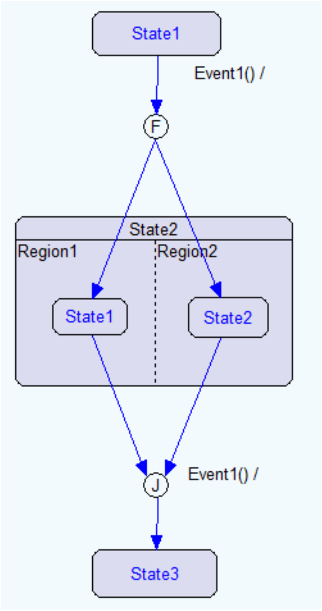


For information about how to draw a transition, see *Creating transitions between your states*, page 130.

DRAWING FORK AND JOIN STATES

Fork and join states are used for going to and from multiple state machines, to and from single state machines, Fork and join states can be used across several state levels. See *Join and fork pseudostates*, page 148.

1. In the Designer, open your project.
2. On the **Diagram** toolbar, click the **Fork State** button ( ). In the diagram, click where you want to insert the fork state.
3. On the **Diagram** toolbar, click the **Join State** button ( ). In the diagram, click where you want to insert the join state.
4. Draw transitions from states to the fork state to a state, and from a state to the join state to states:



DRAWING A JUNCTION STATE

Junction states chain together and split transitions. See *Junction pseudostate*, page 149.

1. In the Designer, open your project.
2. On the **Diagram** toolbar, click the **Junction State** button ( ). In the diagram, click where you want to insert the junction state.
3. Draw transitions to and from the junction state, to and from other states in the diagram.

DRAWING A CHOICE STATE

1. In the Designer, open your project.
2. On the **Diagram** toolbar, click the **Choice State** button ( ). In the diagram, click where you want to insert the choice state.
3. Draw transitions to and from the choice state, to and from other states in the diagram.

For more information, see *Choice state*, page 150.

Working with composite states and regions

What do you want to do?

- *Creating a composite state consisting of concurrent regions*, page 159
- *Hiding the contents in off-page regions*, page 161
- *Adding descriptions for off-page regions*, page 163
- *Excluding states or regions from further processing*, page 163

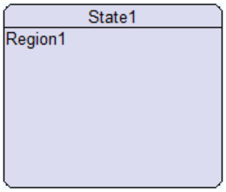
CREATING A COMPOSITE STATE CONSISTING OF CONCURRENT REGIONS

Composite states consist of one or more concurrent regions, where each region contains mutually exclusive states.

You can define regions in states, as well as in the states that represents top-level state machines to define concurrent subsystems and represent hierarchical state machines. See also *State machine hierarchy and concurrency*, page 119.

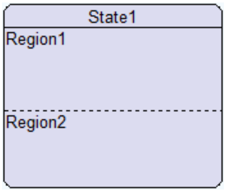
To create a composite state that consists of concurrent regions:

1. In the Designer, open your project.
2. On the **Diagram** toolbar, click the **Composite State** button ( ).
3. In the **Diagram View** window, click to create a state with one region.



4 To deactivate the Composite State tool, right-click.

5 To add a region to the state, right-click anywhere in the region to open the context menu. Select **Insert Region**, and choose where to insert the region.



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 edit the state, right-click in its title area (not in one of the regions) and choose **Edit State** from the context menu. See *Creating a state with a state reaction*, page 153.

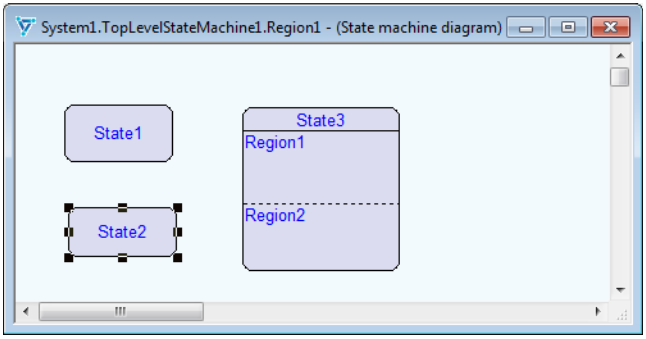
To convert an existing simple state to a composite state:

1 Right-click the simple sate to convert and choose **Insert Region** from the context menu.

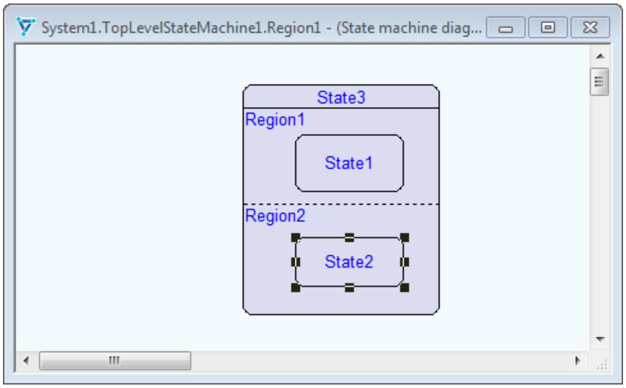
2 The simple state now appears as a composite state.

To insert already created states in a concurrent region:

1. In the state machine diagram, select the state you want to add to a region:



2. Drag the state to the region and drop it.

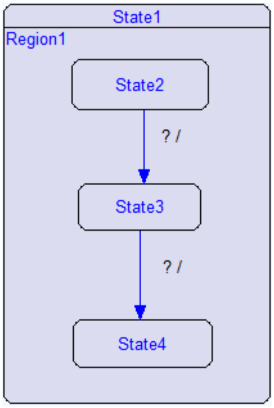


Fill your regions with the required states.

HIDING THE CONTENTS IN OFF-PAGE REGIONS

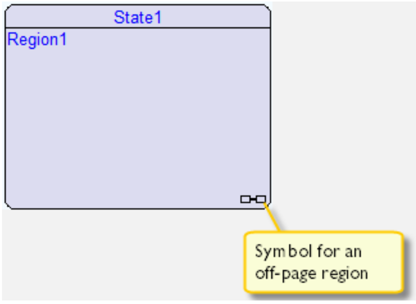
You can choose whether you want to view the contents of a region in the same diagram as the composite state or in a separate diagram. Hiding the contents of a region can give you a better overview of the overall structure of your model if the content of the region is very complex.

1 Assume this region:



2 Right-click the region you want to hide and choose **Off-Page** from the context menu.

The region now appears with a small symbol in the right-bottom corner:

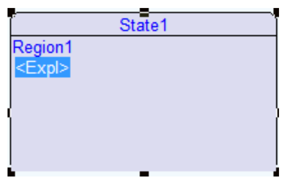


3 To go to the off-page region, click the off-page region symbol or double-click in the region. To return from the off-page region, press the Backspace key.

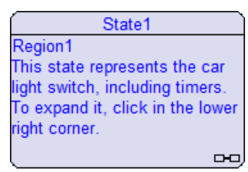
ADDING DESCRIPTIONS FOR OFF-PAGE REGIONS

For off-page regions you an enter a description.

1. Click inside the region in the parent state for the off-page region. An editable field appears:



2. Type a description and press Enter. Your description will appear:



EXCLUDING STATES OR REGIONS FROM FURTHER PROCESSING

Any number of states or regions, on any hierarchical level, can be marked for exclusion from further processing.

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

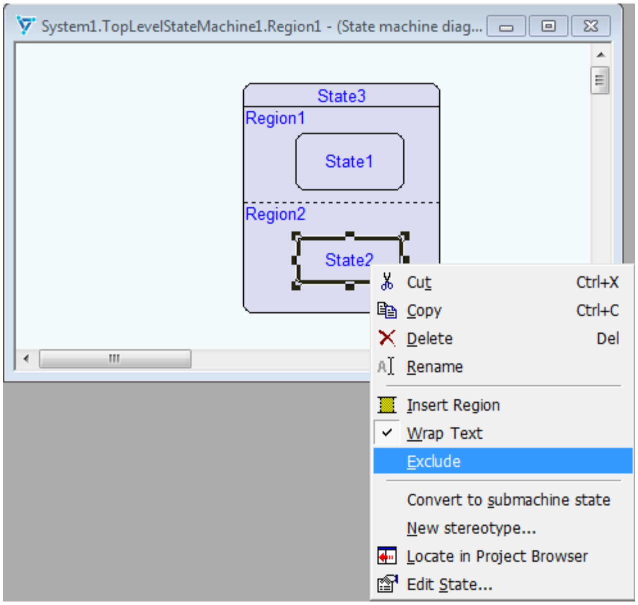
- Configuring your application

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

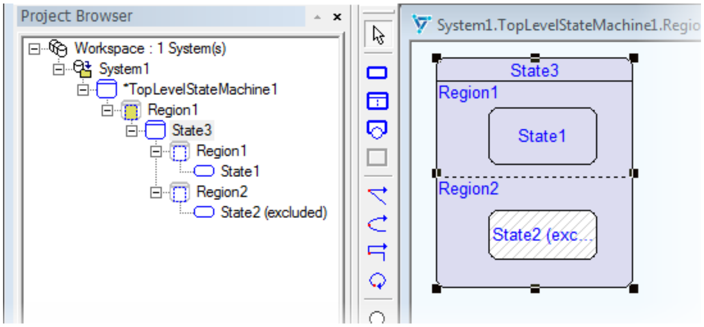
- Adding debug regions to your design to keep track of or detect, for example, error conditions.

  To override exclusion marks, add a separate region where you put your debug state machines and then decide if you want the functionality included in the simulation, in the generated code, or for verification. Verification in particular can greatly benefit from this, letting you, for example, create regions that will enter a dead-end state on certain conditions.

1. Right-click the state or region that you want to exclude and choose **Exclude** from the context menu.



2 Excluded items appear with the tag (excluded) after their name in the state machine diagram and in the **Project Browser** window:



Excluded items also appear with a different background to indicate either that the item is directly excluded itself, or that some parent item higher up the hierarchy has been marked as excluded.

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.

Note that a transition is excluded if it 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.

Transitions

- Introduction to transitions
- Creating transitions

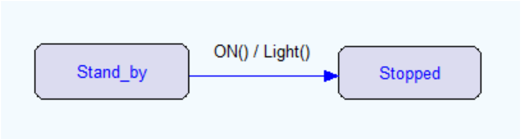
Introduction to transitions

Learn more about:

BRIEFLY ABOUT TRANSITIONS

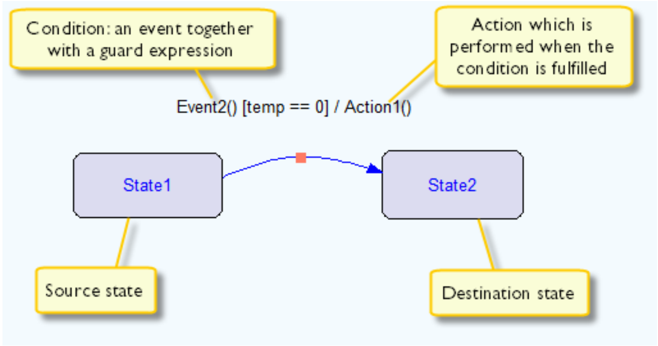
When an event occurs in the state machine environment, the state machine changes its state by performing a *transition*. Optionally, one or more actions can be performed.

For example, when the power is turned on (the event), the state machine changes from the state `Stand_by` to the state `Stopped` and performs the action `Light`:



A transition is defined as a relationship between two states, indicating that a state machine in a specific source state enters a specific destination state when a specified event occurs, provided that specified conditions are satisfied. Any specified actions will then be performed. Formally, this is referred to as a *transition rule*.

Graphically, an ordinary transition is drawn as a solid line that starts from the source state and ends with an arrowhead pointing at the destination state. Beside the line a label is placed, which denotes the condition(s) and optionally an action associated with the transition. The condition is separated from the action by a slash (/). A ? on the condition side illustrates that the transition does not have a trigger, yet. When all conditions are fulfilled, the transition will be triggered and all defined actions will be performed.



To create a transition, first draw the arrow in your state machine diagram and then add your conditions and actions in the **Edit Transition** dialog box. See *Creating transitions*, page 175.

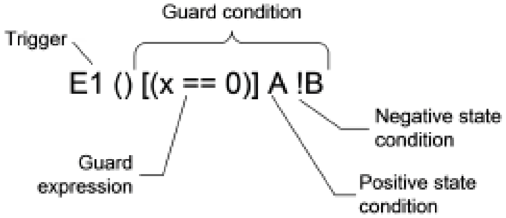
Note: The IAR Visual State concept of transitions is similar to that of UML. In the UML specification, destination state is referred to as *target state*.

In addition to these ordinary transitions, there are special cases of transitions:

- Internal transitions, see *Internal reaction*, page 151
- Completion transitions, see *Completion transitions*, page 173
- Trigger-less transitions, see *Trigger-less transitions*, page 173
- Local transitions, see *Local transitions*, page 173
- Else transitions, see *Else transitions*, page 174.

THE TRANSITION CONDITION

A transition can have one or more conditions associated with it. The conditions must be satisfied for the transition to fire and for the actions to be executed. The condition can consist of a number of parts, for example:



The conditions can consist of these parts:

- *Trigger*. First the trigger must occur; a trigger is what causes the condition to be evaluated. In this example, the expression `E1()`. The trigger can be, for example, an *event*. See *Triggers*, page 169.
- *Guard condition*. When the trigger has occurred, the guard condition must be satisfied for the transition to fire. The guard condition can consist of:
  - *Guard expressions* are expressions that can be evaluated to true or false and which you can create using internal and external variables, action functions, as well as constants and enumerators. In this example, the guard condition consist of this expression: `[(x==0)] A!B`. See *Guard expressions*, page 169.
  - *State conditions* can be either *positive* or *negative*, see *State conditions*, page 170.

To create triggers, guard expressions and state conditions, use the **Transition Elements** dialog box. See *Creating a transition element*, page 184.

Triggers

In Visual State a trigger is what causes the condition of a transition to be evaluated. If the condition is evaluated to be true when a trigger occurs, the transition will fire. Each transition has exactly one trigger.

There are two types of triggers:

- *Explicit triggers*, which can be one of the following:
  - An event, including event parameters. See *Events*, page 179.
  - An event group, which is a collection of events. See *Event group*, page 180.
  - A signal, see *Signal*, page 181.
- *Implicit triggers*, which can be one of the following:
  - Entry (can only be used as a state reaction), see *Entry reaction*, page 151
  - Exit (can only be used as a state reaction), see *Exit reaction*, page 152
  - Completion (can be used in completion transitions), see *Completion transitions*, page 173.

Guard expressions

Guard expressions occur in the transition's condition. For a transition to fire, all guard expressions must evaluate to true.

A guard expression is typically a logical expression or a relational expression. A logical expression is one or more relational expressions separated by logical operators.

A relational expression is composed of a left side, a relational operator, and a right side. The left side is either an external variable, an internal variable, an event parameter, an action function, or a constant or enumerator. The right side can be a complex expression, involving operators and operands of different types.

A guard expression must be legal and must not cause any side effect, or the model might behave incorrectly at runtime if the same guard expression is called more than once.

The guard expression in this example is a logical expression that consists of two relational expressions with the variable x as the left side in both relational expressions:

```
(x >= 0) && (x < 10 + Action(7, 3))
```

See also:

- *Visual State operands, reference information*, page 196 for information about variables, constants, and enumerators
- *Visual State operators, reference information*, page 194
- *Syntax for guard expressions and action expressions*, page 198.

State conditions

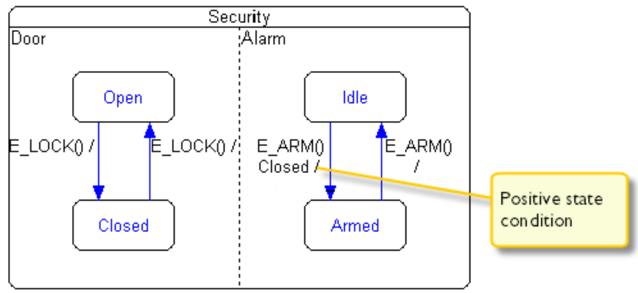
State conditions ensure that another state machine within the same Visual State system satisfies whether another state machine is in a specific state, not in that state, or in a combination of states. State conditions can be used for creating dependencies between one part of the model with another part of the model, that is, to synchronize the parts.

State conditions are part of the transition condition. For a transition to fire, all conditions, including state conditions, must be satisfied.

There are two types of state conditions:

- Positive state conditions

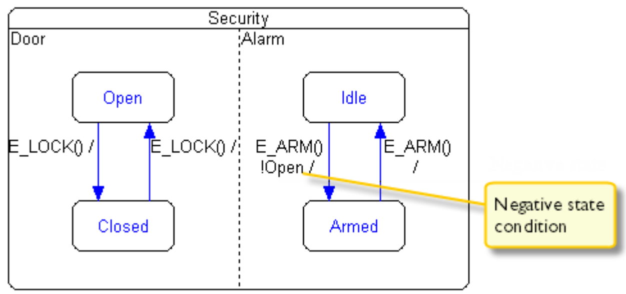
  A positive state condition is when another state machine must be in a specific state for the state condition to be satisfied. Only if the state condition is satisfied, will the transition fire. This figure illustrates an example of a positive state condition in a security system:



Here, the purpose of the state condition is to ensure that the security system will only be armed if all doors are closed. This is achieved by the positive state condition in the state machine `Alarm`. The state condition is specified by the expression `Closed` on the transition from the state `Idle` to the state `Armed`. This transition will only go to the state `Armed` if the state machine `Door` is in the state `Closed`.

- Negative state conditions

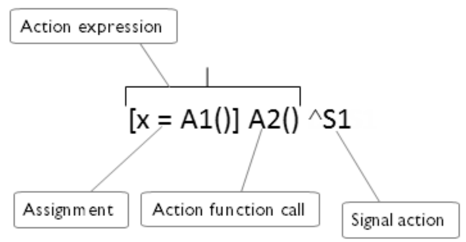
  A negative state condition is the opposite of a positive state condition. The state condition is satisfied only if the state machine is NOT in a specific state. This example illustrates the same security system as in the previous example:



In this case, the state condition is negative in that the door must *not* be in the state `Open`. This is specified by the expression `!Open` on the transition from the state `Idle` to the state `Armed`. This transition will only go to the state `Armed` if the state machine `Door` is not in the state `Open`.

THE TRANSITION ACTION

A transition can have actions that will be executed if the conditions are satisfied (note that a transition action is not mandatory). Like the condition, an action can contain a number of elements, for example:



The action side consists of:

- *Action expressions*—can either be *assignments* of a new value to a variable or stand-alone *action function calls*. In this example, the action expressions are covered by the expression:

  ```
  [x=A1()]A2()
  ```

  See *Assignments in transition actions*, page 172 and *Action function calls*, page 173.
- *Signal actions*—signals will be added to the internal signal queue and will eventually become triggers. See *Signal actions*, page 173.

Assignments in transition actions

Assignments can be used by the action, which means that a variable can be assigned a new value. An assignment contains an assignment operator, in the form of =. On the left side of the assignment operator is the variable that is to be assigned a new value. On the right side of the assignment operator an entire expression can be written, using all allowed unary and binary arithmetic operators, logical operators or bit-manipulation operators, and all allowed operands or values.

This is an example of an allowed assignment:

```
x = A1() + 10
```

See also:

- *Visual State operands, reference information*, page 196 for information about variables, constants, and enumerators
- *Visual State operators, reference information*, page 194
- *Syntax for guard expressions and action expressions*, page 198.

Action function calls

An ordinary function call, which consists of the name of the action function and any argument.

Signal actions

The signal actions are meant to be handled by the internal signal queue and will eventually be treated as triggers. For more information about signals, see *Signal*, page 181. See also *Synchronizing one part of the model with other parts of the model*, page 136 and *Runtime behavior—macrosteps and microsteps*, page 122.

COMPLETION TRANSITIONS

A *completion transition* is a special transition that is used for changing states if a certain state enters a final state.

A transition is said to be a completion transition if it has been marked to use the special trigger `completion` in the **Edit Transition** dialog box.

TRIGGER-LESS TRANSITIONS

Trigger-less transitions are special transitions that do not have an explicit trigger. Such transitions are especially useful in combination with choice states, but they can be used on any transition. See *Choice state*, page 150.

Transitions that are marked as trigger-less are considered to be part of macrosteps, see *Runtime behavior—macrosteps and microsteps*, page 122.

LOCAL TRANSITIONS

Local transitions are special transitions that do not trigger the exit/entry reactions of the source state when they enter a destination substate. This means that there are some constraints:

- A local transition must go from a superstate to a substate.
- If a local transition goes to a pseudostate, any outgoing transition from that pseudostate must also be a local transition.
- A local transition cannot be a self-transition.

To specify a transition as local, select the option **Local transition** in the **Edit Transition** dialog box.

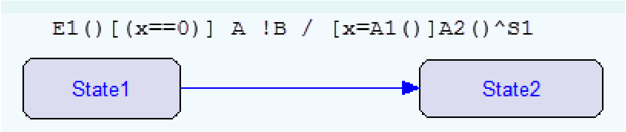
A local transition is drawn with a dashed line in the state diagram.

ELSE TRANSITIONS

An *else* transition means that the transition will only be taken if no other transition leaving the junction or choice state is enabled. This functionality is only available for transitions going out from a junction or a choice state. See *Junction pseudostate*, page 149 and *Choice state*, page 150.

TRANSITION RULE DEDUCTION—AN EXAMPLE

This is an example of a transition condition and action which changes from `State1` to `State2`:



Before the action side can be executed, all conditions on the condition side must be satisfied. Thus, the transition in the figure covers the following:

If

the system is in state `State1`	AND
the event `E1()` occurs	AND
the variable `x` equals 0	AND
the system is in state `A`	AND
the system is in not in state `B`	AND
the system exits state `B` and its children. The exit is performed bottom-up.	AND

then

assign the return value from the action function `A1()` to the variable `x`	AND
execute the action function `A2()`	AND
add the signal `S1` to the signal queue	AND
enter state `State2` and its children. The entry is performed top-down.	

Creating transitions

What do you want to do?

- *Creating transitions between your states*, page 130

See also:

- *Creating a transition element*, page 184.

Transition elements

- Introduction to transition elements
- Working with transition elements and transition element files
- Visual State operators, reference information
- Visual State operands, reference information
- Syntax for guard expressions and action expressions

Introduction to transition elements

Learn more about:

- *Briefly about transition elements*, page 177
- *Events*, page 179
- *Event group*, page 180
- *Signal*, page 181
- *Action function*, page 182

BRIEFLY ABOUT TRANSITION ELEMENTS

The transition elements are building blocks for specifying:

- Transition conditions and actions
- Specific reactions for states.

In other words, transition elements control what should happen and when it should happen for both transitions and states.

To create, define, edit, and delete a transition element, use the **Transition Elements** window. This window also gives a complete overview of the elements created for the project. When you have created a collection of transition elements, you can use them to create your conditions, actions, and state reactions by using the **Edit Transitions** window and the **Edit States** window, respectively.

Transition elements can be stored in *transition element files*—files that contain only transition elements, not states or transitions. These files can be used for organizing the elements in smaller reusable files. For more information, see *Transition element files*, page 179.

These types of transition elements are available:

- Events and event groups—send messages to the state machine, see *Events*, page 179 and *Event group*, page 180.
- Action functions—an activity to be performed by the state machine at a given point in time, see *Action function*, page 182.
- Timer action functions—start timers that cause events, see *Timer action function*, page 183.
- Signal—triggers a transition, like an event, see *Signal*, page 181.
- Primitives—such as internal variables, external variables, constants, and enumerators—express guard conditions and action expressions on transitions, see *Visual State operands, reference information*, page 196.

See also *Creating a transition element*, page 184.

Element declarations and definitions

Every transition element must be designated as either a declaration or a definition. As in the C language, the general rule is that an element can be declared any number of times, but must be defined exactly one time. The only exception to this rule is that action functions and timer action functions can only be declared, but not defined (they are defined externally in user-written code).

Note that although multiple definitions are made, they all define the same single element.

Global and local elements

Transition elements can be either global or local:

- Global transition elements—events, event groups, action functions, timer action functions, external variables, constants, and enumerators—are defined at project level, and have the scope of the project, including all Visual State systems contained in it. The name of a global transition element must be unique within the project.
- Local transition elements—events, event groups, action functions, timer action functions, signals, internal variables, external variables, constants, and enumerators—are defined on top-level state machine level, and have the scope of that state machine itself. External variables are a special case, because they always have scope of the entire project, also when defined on top-level state machine level.

  If two local transition elements are defined with the same name in multiple top-level state machines, they are interpreted as a single element definition having the scope of the parent Visual State system. In such cases, the element definitions must be retyped in exactly the same way in the parallel top-level state machines.

To refer to an element defined at project level or in another top-level state machine, a declaration of the element is required in the top-level state machine that refers the element. For a description of how to add an element defined at project level as a local declaration, see *Declaring global elements locally*, page 185.

Transition elements with overlapping scope cannot have identical names. In addition, names of transition elements of the types external variable and action function must have unique names throughout the entire projects. See *Action function*, page 182 and *External variables*, page 198.

Transition element files

Transition elements can be stored in small, reusable files that contain only transition elements, not states or transitions. You can organize a transition element such as an event or an action function for some specific hardware in its own file, and then reuse that transition element by simply adding the transition element file to another project.

Transition element files work similarly to include files in the C language. You can have as many files as you like, and you can add the same file multiple times.

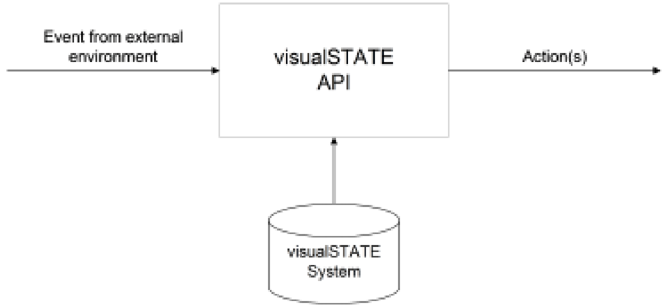
In the Designer, elements from included transition element files are displayed together when you edit transitions, so you can choose elements from any included transition element file.

Note: There can be only one definition of a transition element, but you can have multiple declared transition elements in multiple files. The transition elements work the same way regardless of whether they are organized in transition element files, or in state machine files.

EVENTS

An event is something that happens in the external environment of a Visual State system. In Visual State, events are always processed sequentially. An event causes something to happen, see also *Triggers*, page 169.

The event might cause the transition that it triggers to fire, provided that the conditions on the condition side are satisfied. The transitions that fire will result in one or several actions:



Because an event is considered to be momentary input (such as a button that is pushed), it must be captured and stored before it can be interpreted by IAR Visual State.

Note: You must ensure that the software in the target application is capable of capturing the events required for the Visual State system. Furthermore, you must specify a queue structure if required for the target application.

Event parameters

In IAR Visual State, events can have parameters. An example of a parameter is the activation of a key on a numeric keyboard where the event describes that a key is being passed, and the event parameters describe which key is being pressed.

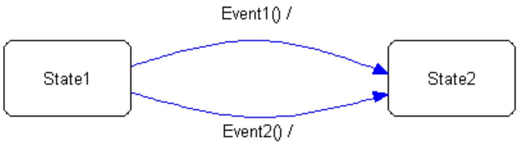
An event parameter can be declared as any of the allowed Visual State data types, and there is no limitation to the number of parameters with which an event can be declared. See also *Visual State data types*, page 197.

EVENT GROUP

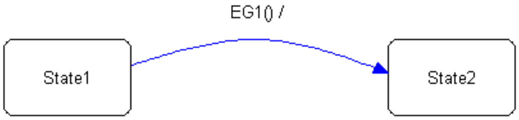
An event group is a collection of disjunct events, and is used as an explicit trigger for a transition. Event groups can be used when several transitions have the same conditions and actions, and only the event varies. Using event groups provides these advantages:

- The state machine model is easier to understand
- Code size is reduced.

If several events can cause a state machine change states, this could be modeled as multiple parallel transitions:



However, if the events that trigger the transitions are included in an event group, you only have to draw one transition. In the following illustration, the event group `EG1` includes the event `Event1` and `Event2` from the previous illustration:



When an event belonging to the event group occurs, it triggers a specific transition, provided that the transition conditions are satisfied. An event group in itself never occurs; only one of its events can occur. An event can be a member of more than one event group.

SIGNAL

Signals trigger transitions, just like events. However, in contrast to an event, which occurs in the external Visual State environment, a signal is sent internally in IAR Visual State. Thus, it functions as an internal trigger for IAR Visual State. That is why signals are allowed in both conditions and the actions of transitions.

A transition can send any number of signals, and because an event can trigger more than one transition, there must be a mechanism for queuing up the signals that have been sent. This queuing is handled entirely by IAR Visual State.

For information about processing an event that triggers transitions with signals, see *Runtime behavior—macrosteps and microsteps*, page 122.

Signal queue

Each Visual State system has a signal queue whose size you must determine. Consider the following issues carefully when you design a Visual State system that includes signals:

- Signal queue size

  Signal queues are drop-if-full FIFO, which means that the queue size is static. You must pre-calculate the required size of the queue and set it so that no overflow will occur. If the system reaches the end of the queue, additional signals will not be added to the queue. However, you can specify that an error code is returned if the signal queue fills up at runtime. Note that the selected signal queue size is checked by the Verificator.

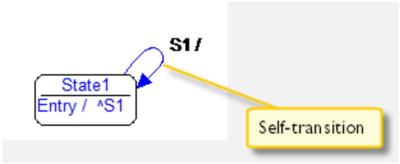
- Priority

  Signals have priority over events and can only be sent internally. It is not possible to send signals from the external environment into the Visual State system.

- Live lock

  Because signals can trigger transitions, which again will send signals, the system can be brought into a live lock. No mechanism is built in to handle this during runtime so you should carefully examine the design to avoid this situation.

  This illustrates the processing of an event that triggers transitions with signals; an example of a live lock:



  The entry reaction of `State1` sends the signal `S1`. This triggers the self-transition, which upon re-entry into `State1` will send signal `S1`, which again will trigger the self-transition, etc. Thus, a live lock has been constructed.

See also *Specifying the signal queue behavior and size*, page 190.

ACTION FUNCTION

An action function is an action that is performed when a transition fires. An action function can be used on a transition in these ways:

- As a function that must be executed when a transition fires. A typical example of this use is calling a device driver.
- As an operand in guard expressions and assignments. In this case, the action function has to return a value.

The activation of action functions is completely handled by IAR Visual State which ensures fully deterministic action function sequencing of the system. However, you must implement the action functions yourself in C source code.

As with ordinary C functions, an action function in IAR Visual State can have parameters and may return a value. For the allowed types of action parameters and action function return values, see *Visual State operands, reference information*, page 196.

See also *Declaring action functions in external C files*, page 192.

Timer action function

Timer action functions start timers that cause an event to occur on timeout. In IAR Visual State, a timer action function has a fixed prototype that contains two arguments, a tick count and an event to be inserted back to IAR Visual State when the timer expires. The return type of timer action functions is `VS_VOID`.

Timer action functions accept two parameters:

- The first parameter is the event to send when the timer times out.
- The second parameter is the number of ticks before a timeout occurs.

Timers are handled outside IAR Visual State and there is no support code provided. If you need a timer, you must include the following in your target application:

- A timer action function for starting a timer, for example from a timer pool, with the specified number of ticks before timeout.
- The event to be sent must be saved.
- On timeout, the timer response function should send the event into the Visual State API to have it processed or to the event queue for later processing.

When a new timer action function is added to a model in the Designer, a new action function with the same name and the suffix `_stop` is automatically added as well. The new timer *stop* function receives the same name as the new timer, and if the timer action function is renamed, the name of the timer stop function changes as well.

If you want the same functionality for existing timer action functions, add timer stop functions manually. If the Validator encounters an action function ending in `_stop`, it checks whether there is a corresponding timer without the `_stop` suffix. For such pairs, the Validator stops the timer when the timer stop function is called as part of the simulation.

To control whether timer stop functions should be created automatically, choose **Tools>Settings>Timer Action** and change the setting.

Working with transition elements and transition element files

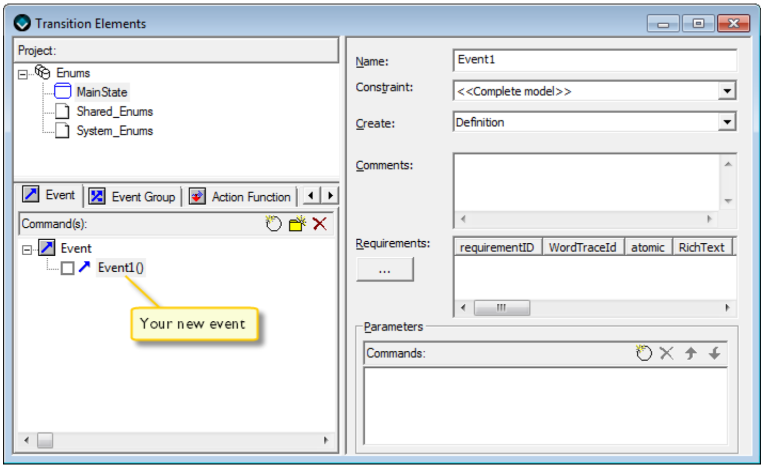
What do you want to do?

- *Creating a transition element*, page 184
- *Making local elements global*, page 185
- *Declaring global elements locally*, page 185
- *Specifying arguments for action function parameters*, page 185

CREATING A TRANSITION ELEMENT

1. In the Designer, open your project.
2. Choose **View>Transition Elements** to open the **Transition Elements** window.
3. In the **Project** pane, select your project in the tree if you want to create a global element. To create a local element, select the state that represents the top-level state machine in the tree.
4. In the **Commands** pane, click the tab that corresponds to the element type for which you want to create an element. For example, click **Event**.
5. On the **Commands** toolbar, click the **New** button ( ).

   A new event with a default name is created in the list:



6 In the right-hand pane, specify an event name and a description for the event, and specify whether it is a definition or a declaration.

You can delete elements by clicking the **Delete** button ( ).

7 For events and action functions, you can specify parameters. Click the **New** button ( ) in the **Parameters** area. Select the parameter and choose a parameter type from the drop-down list. See also *Visual State data types*, page 197.

To delete parameters for events and action functions, click the **Delete** button.

MAKING LOCAL ELEMENTS GLOBAL

To make a previously created local element global, follow these steps:

1 In the **Project** pane of the **Transition Elements** window, select the state that represents the top-level state machine in the tree where the element you want to make global was created previously.

2 In the **Commands** pane, click the tab that corresponds to the correct element type. For example, click **Event**.

3 Select the element in the **Commands** area and drag it from there to the project in the **Project** pane.

DECLARING GLOBAL ELEMENTS LOCALLY

A global transition element that is defined in the project must be added as a declaration in the files where you want to use it. Follow these steps:

1 In the **Project** pane of the **Transition Elements** window, select your project in the tree.

2 In the **Commands** pane, click the tab that corresponds to the correct element type. For example, click **Event**.

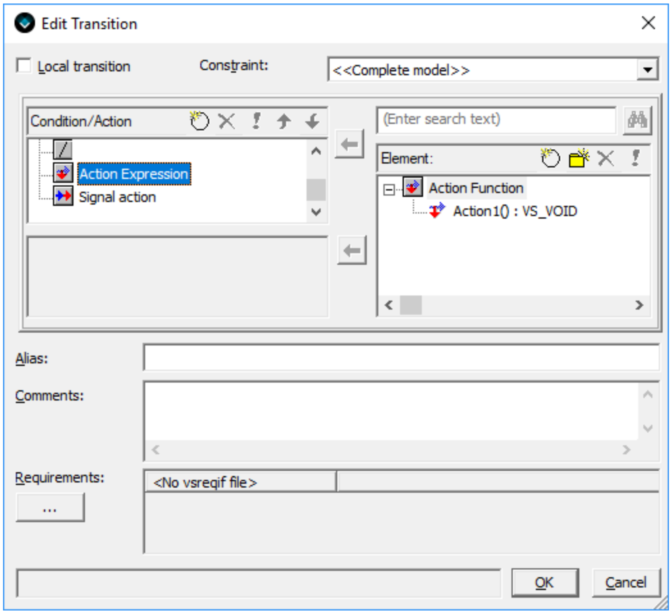
3 Select the element in the **Commands** area and Ctrl+drag it from there to the file in the **Project** pane where you want to add a local declaration of it.

SPECIFYING ARGUMENTS FOR ACTION FUNCTION PARAMETERS

If you have defined an action function that takes parameters, you can specify the arguments in the individual transitions and state reactions where the action function is used, as follows.

1 In the state machine diagram, double-click the state or the transition label (condition and action).

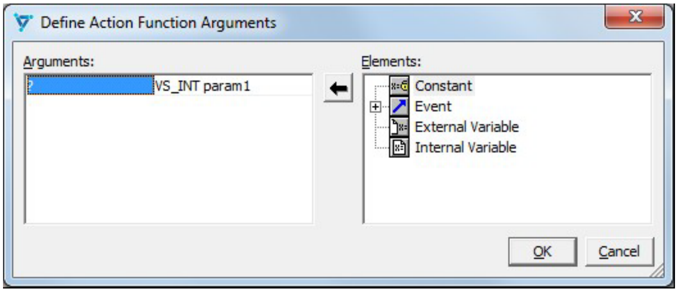
The **Edit State** or **Edit Transition** dialog box, respectively, is displayed. In this example, the latter will be used, but the same procedure applies to the **Edit State** dialog box.



2 In the **Condition/Action** area, select **Action Expression** to display the valid transition elements in the **Element** area.

3 In the **Element** pane, select the transition element to add, for example the action function, and double-click it (or click the Left Arrow button). The element will be added to the **Condition/Action** area.

4 If the action function you just added can take arguments, double-click it. The **Define Action Function Arguments** dialog box is displayed.



5 Double-click a transition element in the **Elements** area to expand the tree. Select the item that you want to use as argument and double-click it.

6 Specify the name of the argument and its type, see *Visual State data types*, page 197.

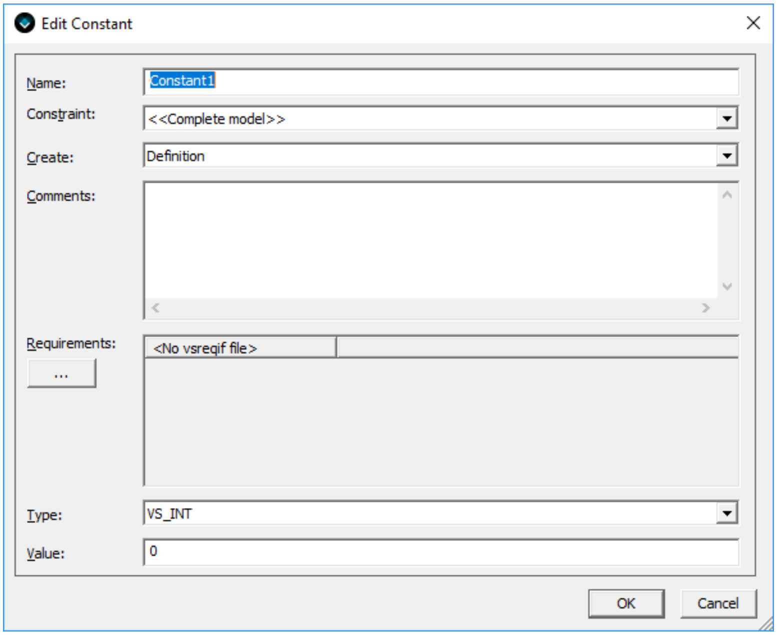
ADDING ASSIGNMENTS AND GUARD EXPRESSIONS

You can add assignments and guard expressions to your transition condition and action using the **Edit State** and **Edit Transition** dialog boxes, respectively. This example shows how to add guard expressions and assignments to a state reaction, but the same procedure applies to transitions.

1 In the state machine diagram, double-click the state for which you want to add an assignment or guard expression to open the **Edit State** dialog box.

2 Click the appropriate tab and select **Guard Expression** or **Action Expression**, respectively. Any defined elements are listed. Otherwise, if you need a new action expression or guard expression, click the **New** button ( ).

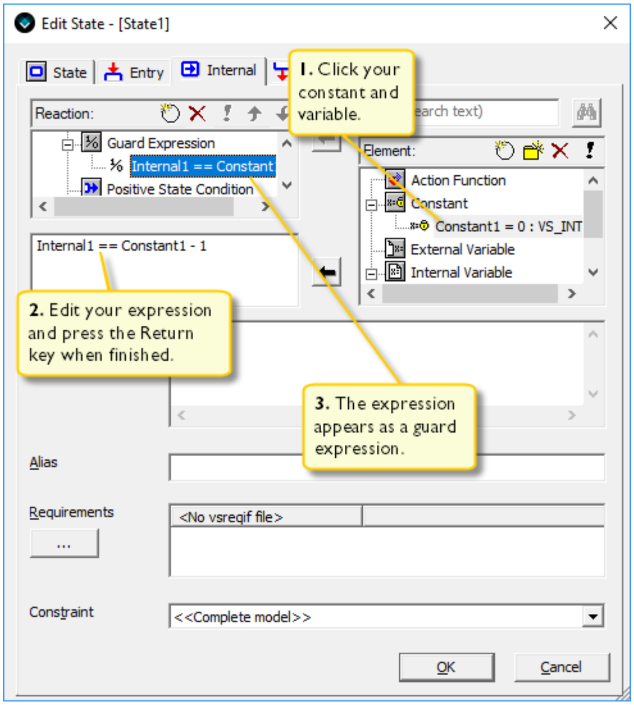
3 In the **Element** area, select the item you want to apply, for example **Constant**, and click the **New** button ( ). The **Edit Constant** dialog box is displayed.



Specify the details of the constant and click **OK**.

4 Repeat the previous step to create a second element, for example an internal variable.

5 To create a guard expression, double-click your newly created constant and variable in the **Element** area. They will appear in the text field under the **Reaction** area. Edit the expression according to the syntax. See *Syntax for guard expressions and action expressions*, page 198.



6 Press the Enter key, or click another item to accept the value.

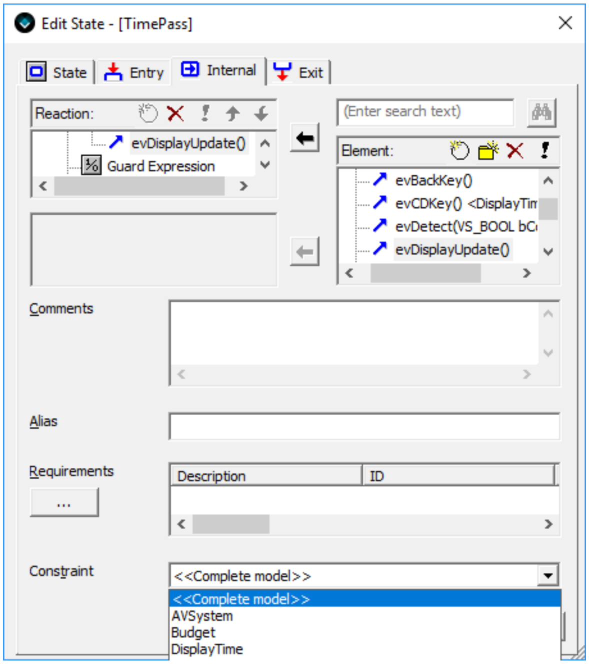
SETTING A CONSTRAINT FOR A STATE REACTION

When you have a state reaction—an internal reaction, entry reaction, or exit reaction—you can set a constraint on it.

1 In the Designer, open your project.

2 In the state machine diagram, double-click the state that contains the reaction that you want to modify. The **Edit State** dialog box is displayed. Alternatively, right-click the state and choose **Edit State** from the context menu.

3 Select the reaction you want to modify on the **Entry**, **Internal**, or **Exit** tab, and select a **Constraint** in the list at the bottom. The constraint will be applied to the reaction.



4 For every internal, entry, or exit reaction, you can choose one constraint, or select `<<Complete model>>` if the reaction should have no constraint.

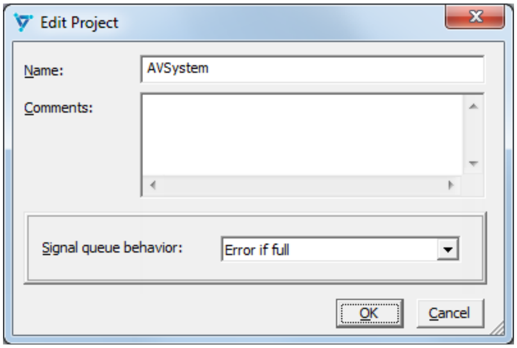
For more information, see *Include/exclude parts in a variant*, page 218.

SPECIFYING THE SIGNAL QUEUE BEHAVIOR AND SIZE

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

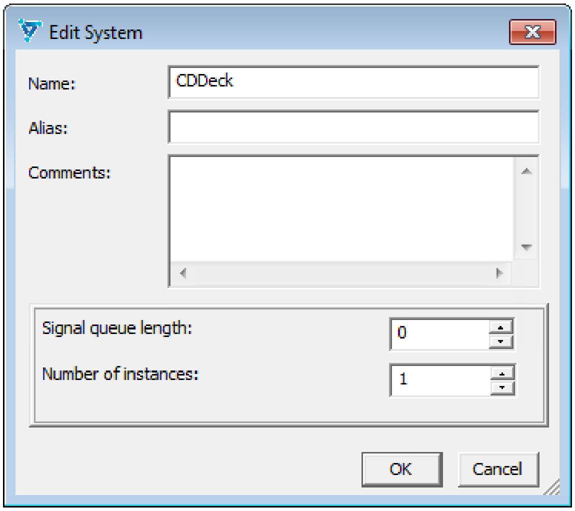
1 In the Designer, open your project.

2 In the **Project Browser** window, right-click the project and choose **Edit**. The **Edit Project** dialog box is displayed:



Specify the signal queue behavior, either **Drop if full** or **Error if full**.

3 In the **Project Browser** window, right-click the appropriate system and choose **Edit**. The **Edit System** dialog box is displayed:



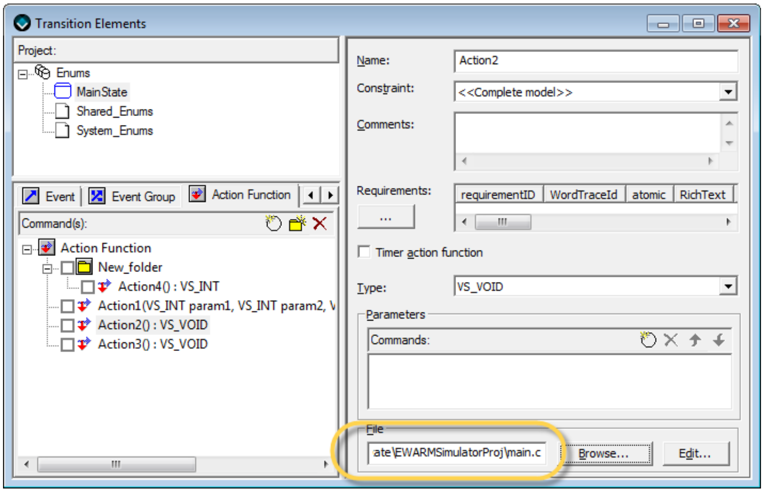
Specify the behavior—the length and number of instances. See *Signal queue*, page 181.

DECLARING ACTION FUNCTIONS IN EXTERNAL C FILES

Action functions can be declared and implemented in an external C file.

1 Choose **View>Transition Elements** to open the **Transition Elements** window. Click the **Action Function** tab and select the action function for which you want to use a C declaration file.

2 In the **File** area, use the **Browse** button to locate a C file to use. The filename is displayed:



3 Click the **Edit** button to open the C file for editing. The editor that you have specified on the **Tools>Settings>External Editor** page will be started. Edit and save the file. Return to the Designer when you are finished editing.

See also *Setting up an external editor for action functions*, page 192.

SETTING UP AN EXTERNAL EDITOR FOR ACTION FUNCTIONS

You can use another editor than the default editor.

1 Choose **Tools>Settings** to open the **Settings** dialog box.

2 On the **External Editor** page, specify an editor of your choice.

SEARCHING FOR A TRANSITION ELEMENT

You can search for a specific transition element to find out in which transitions and state reactions of the model it is used.

1. In the Designer, open your project.
2. Click the **Find** button on the **Standard** toolbar, or choose **Edit>Find**.
3. In the **Find** dialog box, specify your search criteria and click **Find** to start the search

   The result of the search is displayed in the output window (**Find** tab). An icon shows where the element was found, and a description is given. To jump to the found element, either double-click the result or press F4 to browse find locations.

   See also *Find dialog box*, page 276.

CREATING AND ADDING A NEW TRANSITION ELEMENT FILE

1. In the Designer, open your project.
2. Choose **View>Project Browser** to open the **Project Browser** window. Select the **File View** tab.
3. Right-click on the file that you want to add an element file to, and choose **Add New Element File** from the context menu.
4. Double-click the new element file (`.vste`) to open the **Transition Elements** window. You might have to expand the view to see the newly created file.
5. Define the elements that you want the new file to include, following the instructions in *Creating a transition element*, page 184, and related tasks.

ADDING AN EXISTING TRANSITION ELEMENT FILE

1. In the Designer, open your project.
2. Choose **View>Project Browser** to open the **Project Browser** window. Select the **File View** tab.
3. Right-click on the file that you want to add an element file to, and choose **Add Existing Element File** from the context menu.
4. Navigate to the element file (`.vste`) to add it. You might have to expand the view to see the newly added file.

EDITING THE CONTENTS OF A TRANSITION ELEMENT FILE

1. In the Designer, open your project.
2. Choose **View>Transition Elements** to open the **Transition Elements** window.

3 In the **Project** pane, select the transition element file that you want to edit.

4 Edit the elements, following the instructions elsewhere in this documentation.

DELETING, RENAMING, OR SAVING A TRANSITION ELEMENT FILE UNDER A NEW NAME

1 In the Designer, open your project.

2 Choose **View>Project Browser** to open the **Project Browser** window. Select the **File View** tab.

3 Right-click on the file that you want to change. Use the commands on the context menu to delete, rename, or save the file under a new name.

Visual State operators, reference information

Reference information about:

- *Precedence of operators*, page 194
- *Assignment operators*, page 194
- *Binary arithmetic operators*, page 195
- *Bit manipulation operators*, page 195
- *Logical operators*, page 195
- *Relational operators*, page 195
- *Unary arithmetic operators*, page 196
- *Unary bitwise operators*, page 196
- *Unary logical operators*, page 196

PRECEDENCE OF OPERATORS

The precedence of operators is according to Standard C.

ASSIGNMENT OPERATORS

This is the assignment operator:

Operator	Symbol	Format	Operation
Assignment	=	`x = y`	Assign value of y to x.

Table 7: Assignment operators

BINARY ARITHMETIC OPERATORS

These are the binary arithmetic operators:

Operator	Symbol	Format	Operation
Multiplication	*	`x * y`	x times y.
Division	/	`x / y`	x divided by y.
Modulus	%	`x % y`	Remainder of x divided by y.
Addition	+	`x + y`	x plus y.
Subtraction	-	`x - y`	x minus y.

Table 8: Binary arithmetic operators

BIT MANIPULATION OPERATORS

These are the bit manipulation operators:

<table>
<tr><th>Operator</th><th>Symbol</th><th>Format</th><th>Operation</th></tr>
<tr><td>Right shift</td><td>>></td><td>x >> y</td><td>x shifted right y bits.</td></tr>
<tr><td>Left shift</td><td><<</td><td>x << y</td><td>x shifted left y bits.</td></tr>
<tr><td>Bitwise AND</td><td>&</td><td>x & y</td><td>x bitwise ANDed with y.</td></tr>
<tr><td>Bitwise inclusive OR</td><td>|</td><td>x | y</td><td>x bitwise ORed with y.</td></tr>
<tr><td>Bitwise exclusive OR (XOR)</td><td>^</td><td>x ^ y</td><td>x bitwise exclusive ORed with y.</td></tr>
</table>

Table 9: Bit manipulation operators

LOGICAL OPERATORS

These are the logical operators:

<table>
<tr><th>Operator</th><th>Symbol</th><th>Format</th><th>Operation</th></tr>
<tr><td>Logical AND</td><td>&&</td><td>x && y</td><td>1 if x and y are nonzero, else 0.</td></tr>
<tr><td>Logical OR</td><td>||</td><td>x || y</td><td>1 if x or y are nonzero, else 0.</td></tr>
</table>

Table 10: Logical operators

RELATIONAL OPERATORS

These are the relational operators:

Operator	Symbol	Format	Operation
Greater than	>	`x > y`	1 if x is greater than y, else 0.
Less than	<	`x < y`	1 if x is less than y, else 0.

Table 11: Relational operators

Operator	Symbol	Format	Operation
Greater than or equal to	`>=`	`x >= y`	1 if x is greater than or equal to y, else 0.
Less than or equal to	`<=`	`x <= y`	1 if x is less than or equal to y, else 0.
Equal to	`==`	`x == y`	1 if x is equal to y, else 0.
Not equal to	`!=`	`x != y`	1 if x is not equal to y, else 0.

Table 11: Relational operators

UNARY ARITHMETIC OPERATORS

These are the unary arithmetic operators:

Operator	Symbol	Format	Operation
Unary minus	`-`	`-x`	Negated value of x.
Unary plus	`+`	`+x`	Value of operand.

Table 12: Unary arithmetic operators

UNARY BITWISE OPERATORS

This is the unary bitwise operators:

Operator	Symbol	Format	Operation
Bitwise complement	`~`	`~x`	Bitwise complement of x.

Table 13: Unary bitwise operators

UNARY LOGICAL OPERATORS

This is the unary logical operators:

Operator	Symbol	Format	Operation
Logical negation	`!`	`!x`	1 if x is zero, else 0.

Table 14: Unary logical operators

Visual State operands, reference information

Reference information about:

- *Visual State data types*, page 197
- *Internal variables*, page 198
- *External variables*, page 198
- *Visual State constants*, page 198
- *Visual State enumerations*, page 198

VISUAL STATE DATA TYPES

These are the Visual State data types and their representations:

Visual State data type	Range (min/max)	Code size (bits)	Representation ‡)
VS_VOID †)	—	—	void
VS_VOIDPTR †)	—	32*)	void *
VS_BOOL	0 / 255	8 *)	unsigned char
VS_UCHAR	0 / 255	8 *)	unsigned char
VS_SCHAR	-128 / 127	8 *)	signed char
VS_INT	-2147483648 / 2147483647	32 *)	int
VS_INT8	-128 / 127	>=8	**)
VS_INT16	-32768 / 32768	>=16	**)
VS_INT32	-2147483648 / 2147483647	>=32	**)
VS_UINT	0 / 4294967295	32*)	unsigned int
VS_UINT8	0 / 255	>=8	**)
VS_UINT16	0 / 65535	>=16	**)
VS_UINT32	0 / 4294967295	>=32	**)
VS_FLOAT	±3.402823466e+38	32*)	float
VS_DOUBLE	±1.7976931348623158e+308	64*)	double
VS_EVENT_TYPE	n/a	>=8	**)

Table 15: Visual State data types

*) The code size on target depends on the compiler. The figures specified in the table are those used by the Validator.

**) Will be code-generated as the smallest ordinal (integer) type capable of representing the number of events in the model.

If you are using a target where the integer size is 16 bits, make sure to use values that do not exceed that size. Otherwise, you might get different behavior when running your application in the Validator compared to running it on the target.

†)Not allowed for internal variables, external variables, or action function return variables.

‡) The listed representation reflects C89. To specify another representation, for example C99, choose **Project>Options>Code generate** in the Navigator. In the left pane, select the project and click the **Types** tab. Change the **Types style** option.

INTERNAL VARIABLES

The scope of the internal variables is the system. Internal variables cannot be referenced outside the system. An internal variable can be declared as any of the allowed Visual State data types, except the type `VS_VOID` and `VS_VOIDPTR`, see *Visual State data types*, page 197. IAR Visual State supports arrays of internal variables.

If there are multiple instances of the system, every instance will get its copies of the internal variables.

EXTERNAL VARIABLES

External variables are variables that can be used both inside the state machine model and in the user-written code.

An external variable can be declared as any of the allowed Visual State data types, except the types `VS_VOID` and `VS_VOIDPTR`, see *Visual State data types*, page 197.

IAR Visual State supports arrays of external variables.

VISUAL STATE CONSTANTS

Visual State constants are similar to Standard C macros. In IAR Visual State, constants must have an explicit type.

Examples of Coder-generated constants:

```
#define MAX_SPEED ((VS_INT)100)
#define PI_AS_FLOAT ((VS_FLOAT)3.1415f)
#define PI_AS_DOUBLE ((VS_DOUBLE)3.1415)
```

VISUAL STATE ENUMERATIONS

Visual State enumerations are similar to Standard C enumerations. The enumerators of an IAR Visual State enumeration will always have the implicit type `int` (`VS_INT`). The ordinal numbers are not assignable but will have values from 0 to N-1, where N is the number of enumerators in the enumeration. Enumerations are used in a similar way (and in some cases exactly the same way) to constants in most parts of Visual State. Enumerations can only be defined and located in an Element file. To hold values from an enumeration, variables should preferably be of type `int` (the user-defined `enumeration` type is not available inside Visual State).

Syntax for guard expressions and action expressions

A Visual State expression can be any valid C expression of the C operators, identifiers, floating-point and integer constants, but with some limitations.

These operators are not supported:

- . (member)
- -> (member by pointer)
- * (indirection (dereference))
- `(type)` (cast)
- `sizeof`
- ?: (ternary)
- , (comma)

These elements of the C language are not supported either:

- `long double` floating-point constants
- suffixes for integer constants
- octal integer constants
- hexadecimal integer constants
- multiple assignments or increments/decrements in the same expression.

These limitations also apply:

- An assignment and increment/decrement operator must be placed in the front of the expression, not in the middle. In other words, `a(b = c)` is not supported.
- Event arguments of `void*` type can only be passed to action function arguments.
- If you want to verify your state machine model using the Verificator, some additional limitations apply. See *Non-verifiable elements*, page 417.

Reusing designs using state machine templates

- Introduction to state machine templates
- Working with state machine templates and submachine states

Introduction to state machine templates

Learn more about:

- *State machine templates and submachine states*, page 201
- *Hints for designing state machine templates*, page 203

STATE MACHINE TEMPLATES AND SUBMACHINE STATES

A *state machine template* represents a design that you can reuse, in projects, systems, within the same top-level state machine, or even in another state machine template. To instantiate the template, you create a *submachine state* that you associate with the template.

This is more flexible than using the functionality provided by top-level state machines. Because the design in a state machine template can be flexible so that you do not have to configure the actual behavior for it until you associate it with a submachine state. If you want, you can also design your state machine template so it will behave identically in all cases.

For the template, you specify required transition elements, and later, when you want to instantiate the template, you *bind* the abstract transition elements of the template with the actual transition elements of the submachine state.

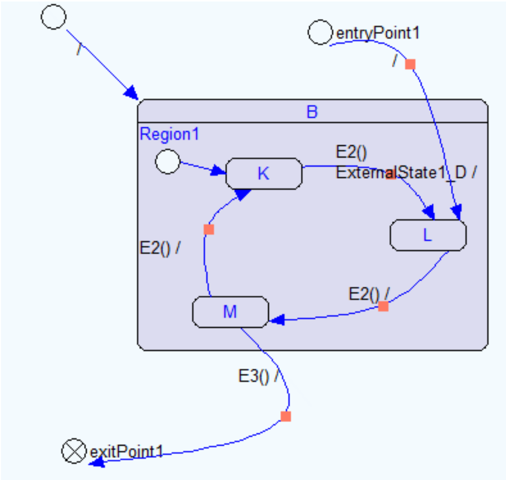
You can also customize the template by specifying an *entry* and *exit* point state. In addition to the default state, entry states define the possible states where the template can be entered. The exit states define the designed ways for leaving the template. When you instantiate the template, corresponding *entry and exit point references* are automatically created on the submachine state; drawn like small circles on the state frame. Likewise, deleting an exit/entry point state from the template, will automatically delete the corresponding entry/exit point references (and any transitions).

If any transitions in your state machine template has state conditions (state synchronizations), you must bind these with other states—either states internally in the template or externally outside the template.

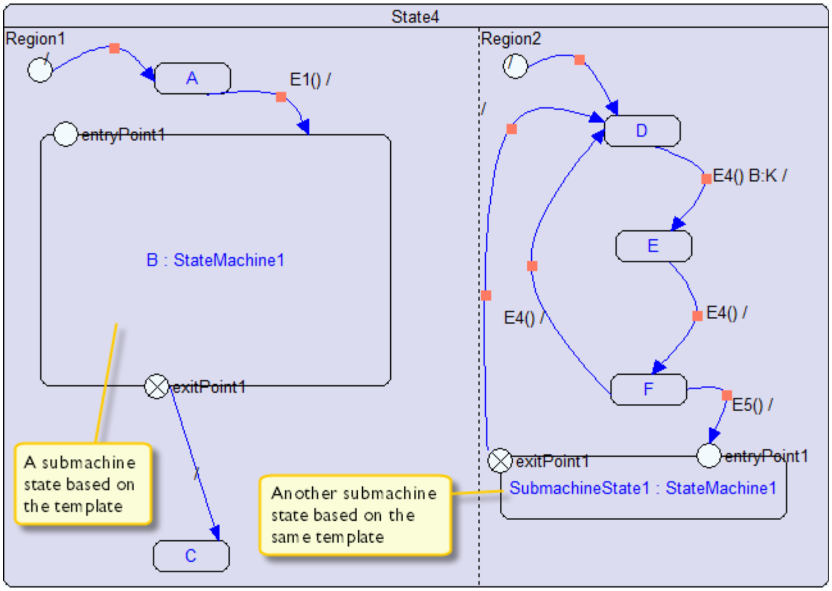
Thus, to instantiate your template, you must:

- Create a submachine state and associate it with the template
- Bind the abstract transition elements of the template with the actual transition elements of the model where your submachine state has been added.

This is an example state machine model for which there has been created a template:



This example shows a state machine model where two submachine states are instances of the template, The example also shows the entry and exit states reference points:

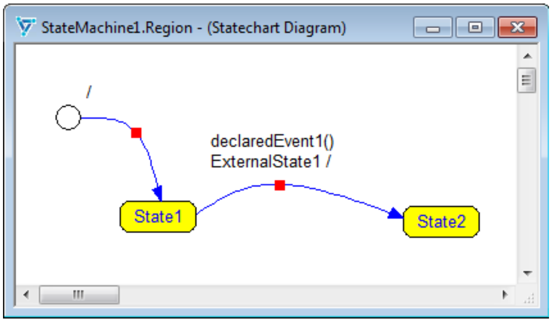


A state machine template is saved in its own file (`vssm`), which means that you can keep track of the file in your version control system.

HINTS FOR DESIGNING STATE MACHINE TEMPLATES

When designing a state machine model that you want to create a template for, you can tailor the template so that the template itself does not know about the runtime event, action, constant, enumerator, signal, variable, or state that is part of a transition. In the state machine template, you can choose to add transition elements by using the **Transition Element** window, and set the type to either Definition or Declaration. Any transition element that you add as a definition will be the same in all instances of the state machine template, whereas transition elements that are declarations can be bound to an actual element for each instance of the state machine template.

This example has a transition that uses a trigger `declaredEvent1`, and one positive synchronization state `ExternalState1`.



The trigger `declaredEvent1` is an event added as an declaration for the state machine template. This lets you bind the event and the synchronization state to use for an instance of the state machine template, for each use of the state machine template from a submachine state.

The synchronization state, `ExternalState1`, was added using the **Edit Transition** window. See also *Binding state conditions*, page 214 and *Edit State dialog box*, page 261.

Working with state machine templates and submachine states

What do you want to do?

- *Creating state machine templates*, page 204
- *Instantiating a state machine template*, page 207
- *Drawing an entry (exit) point state*, page 210
- *Binding state conditions*, page 214

CREATING STATE MACHINE TEMPLATES

There are three ways to create a state machine template in your project:

- Creating a new state machine template
- Adding an existing state machine template from another project
- Converting an existing state to a submachine state that is associated with a state machine template

  Typically, this method is useful when you have designed a composite state and realize that you want to reuse it.

To create a new empty state machine template:

1 Right-click the project or an existing state machine in the **Project Browser** window and choose **Add New State Machine Template** from the context menu.

2 This adds a new empty state machine template that is not yet associated with any submachine state.

To use an existing state machine template:

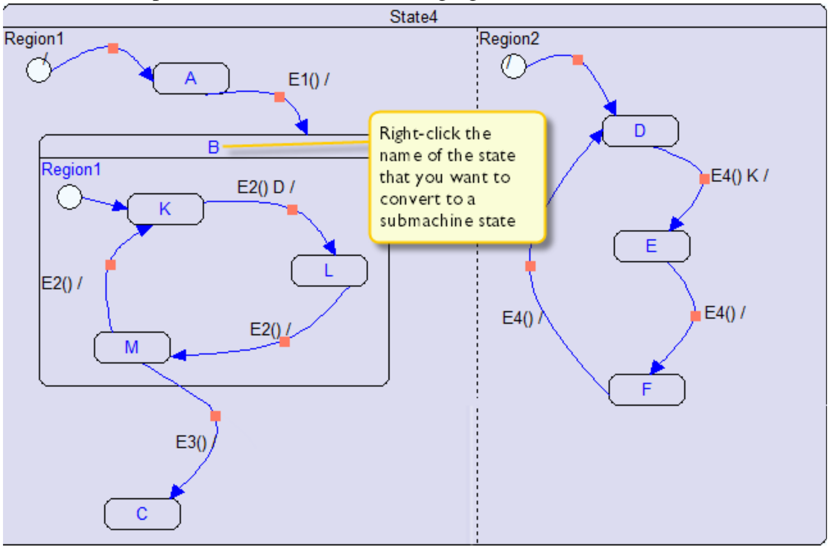
1 Right-click the project in the **Project Browser** window and choose **Add Existing State Machine Templates** from the context menu.

2 Use the dialog box to locate the state machine template file for the template you want to add to your project.

To convert a state to a submachine state that is associated with a state machine template:

1 Design a normal hierarchic state with the behavior you need, as part of another state hierarchy.

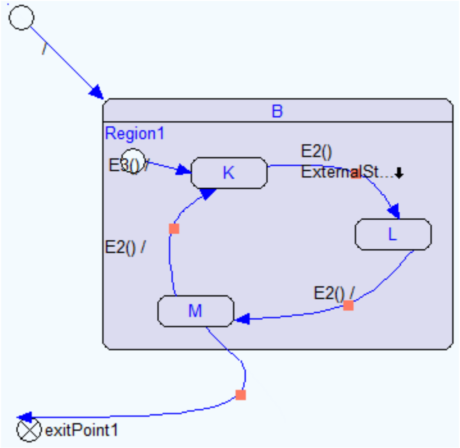
This is an example state machine before changing a state to a submachine state:



2 Right-click the name of the state that you want to convert to a submachine state (in this example, `B`) and choose **Convert to submachine state** from the context menu.

3 The state has now been converted to a submachine state and the contents of the original state is now a new free-standing state machine template as part of the project reusable in other places, and visible in the **Project Browser** window.

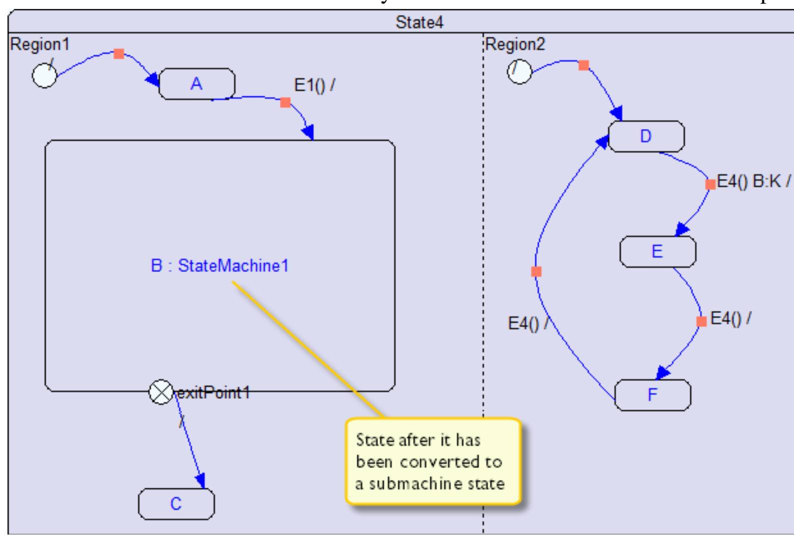
This figure shows the automatically created state machine template based on the original state `B`:



In this example, the outgoing transition from state `M` to state `C` (which crosses the state boundary) has been converted to transition going to an exit connection point inside the state machine template.

The names of the transition elements in the template will be inherited from the original state machine model. Because you might want to instantiate the template several times in your state machine model, you should consider renaming the transition elements in the template, to give them more abstract names.

This figure shows state B after it has been converted to a submachine state; the submachine state has been automatically associated with the state machine template:



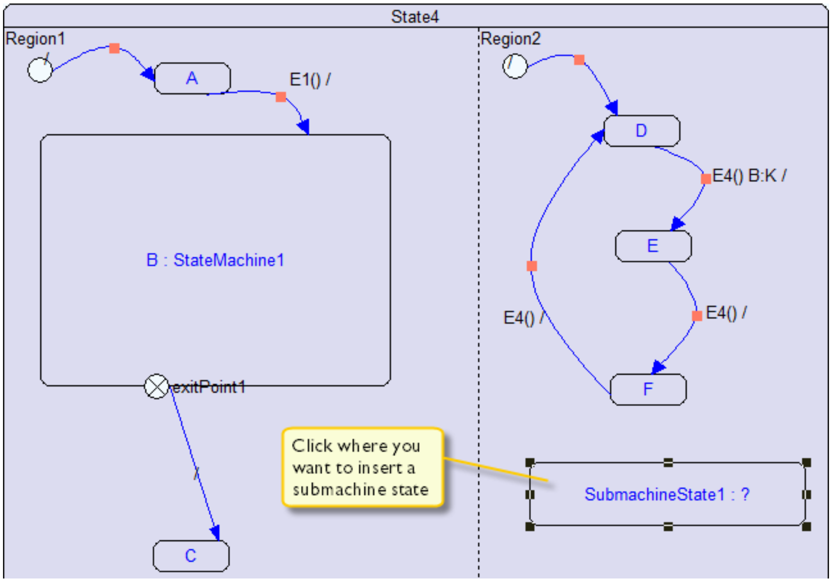
A exit reference point has been created on the submachine state frame for the outgoing transition to state C. Any transition element (event, action, etc) that was used in the converted state was added as a declared transition element in the created state machine template, with appropriate bindings being set in the submachine state. The behavior of the system before and after converting a state to a submachine state is the same.

INSTANTIATING A STATE MACHINE TEMPLATE

1 In the Designer, open your project.

2 On the **Diagram** toolbar, click the **Submachine State** button (). In the state machine diagram below a state machine, click where you want to insert the submachine state (or in other words, click where you want to instantiate a template).

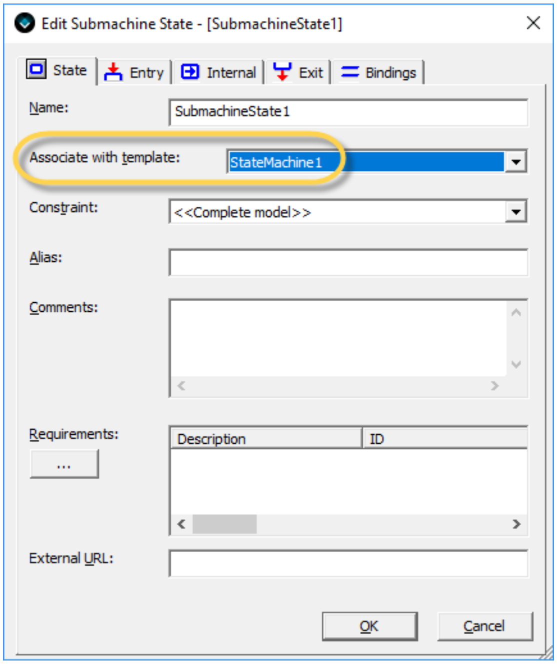
For example:



3 To associate the submachine state with the state machine template, right-click the newly created submachine state and choose **Edit Submachine State** from the context menu.

The **Edit Submachine State** dialog bow is displayed and it works like the **Edit State** dialog box, but has the extra text field **Associate with template** and the extra tab **Bindings**. For reference information, see *Edit Submachine State dialog box*, page 265.

4 Use the **Associate with template** drop-down list to specify the template that you want the submachine state to be associated with, for example `StateMachine1`.

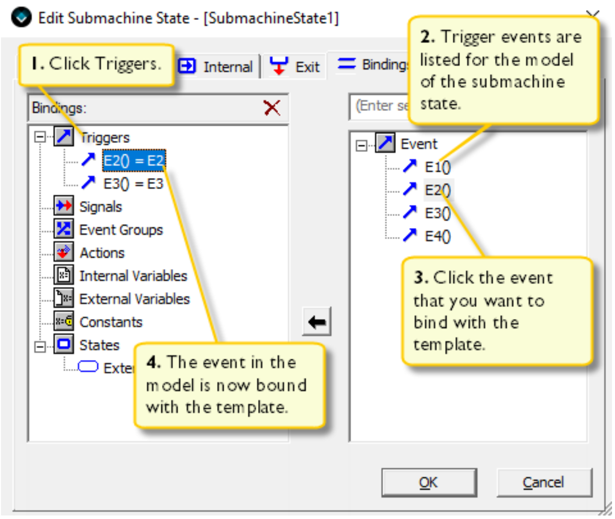


5 Click the **Bindings** tab.

The **Bindings** area lists the elements available in the template. You should now bind the transition elements in template with the counterparts in the model where the submachine state has been added.

6 To bind, for example your triggers, click **Triggers** in the **Reaction** area.

The triggers available in the model represented by the submachine state appears in the **Elements** area. Select the trigger in the Reaction area, for example `E2`, and double-click the actual trigger in the Element pane, `E2`. The dialog box now looks like this:



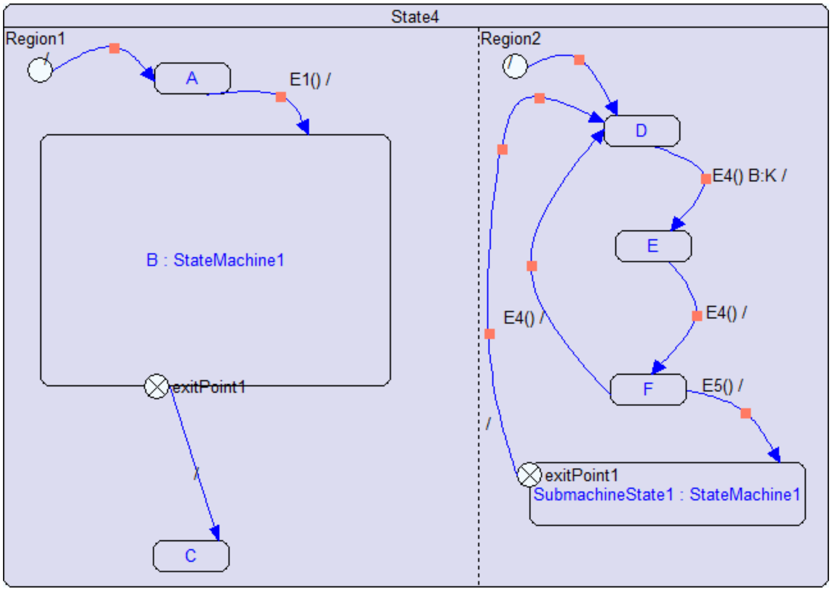
Now the state machine template is instantiated in two places—two different instances. They do not work the same, because they have different bindings. So even though the model uses the same template in two places, the two instances work differently.

As a next step, you must set up the entry and exit point states, see *Drawing an entry (exit) point state*, page 210.

DRAWING AN ENTRY (EXIT) POINT STATE

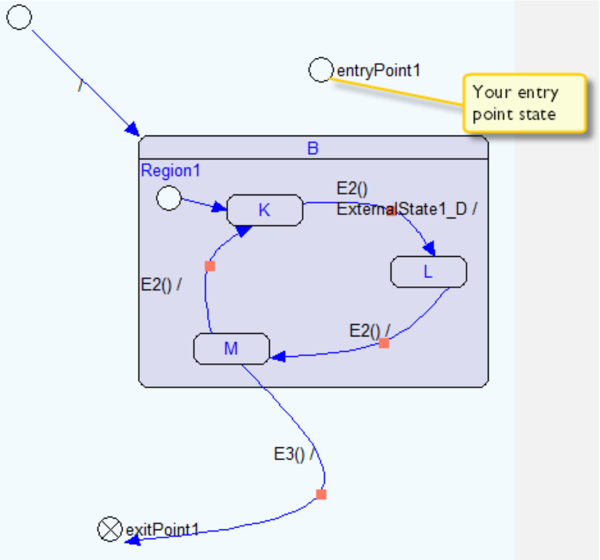
A state machine template will generally have entry and exit point states. A submachine state that is instantiated from a state machine template inherits all entry and exit point states from the template.

For this example, you need at least one transition that enters the submachine state, and potentially also at least one transition leaving the state via the exit point reference, which means it might look like this:

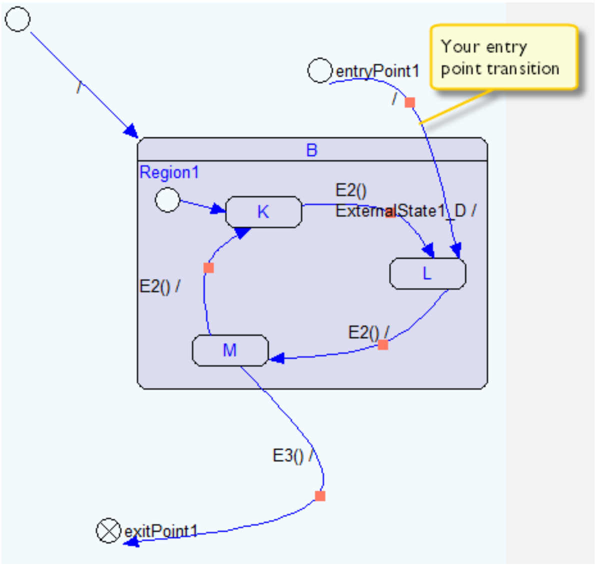


- In the Designer, open your project.

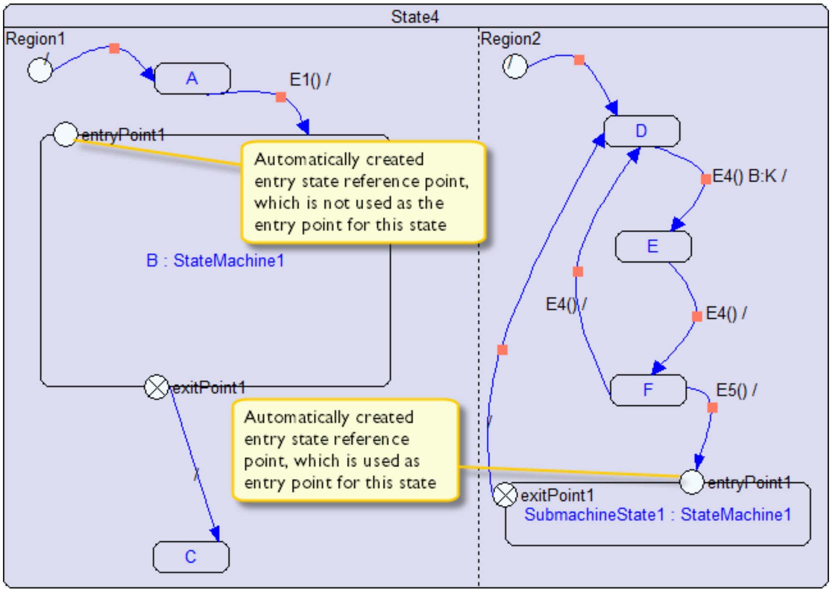
2 On the **Diagram** toolbar, click the **Entry (Exit) Point State** button (). In your template state machine, click where you want to insert the entry (exit) point state.



3 Draw a transition from the entry point state to some other state in the diagram.



When you add an entry (exit) point state to a state machine template, a corresponding entry (exit) state reference point will be automatically added to the frame of the corresponding submachine states.



In this example, the entry point reference state is used as entry point in one of the template instances, but not in the other.

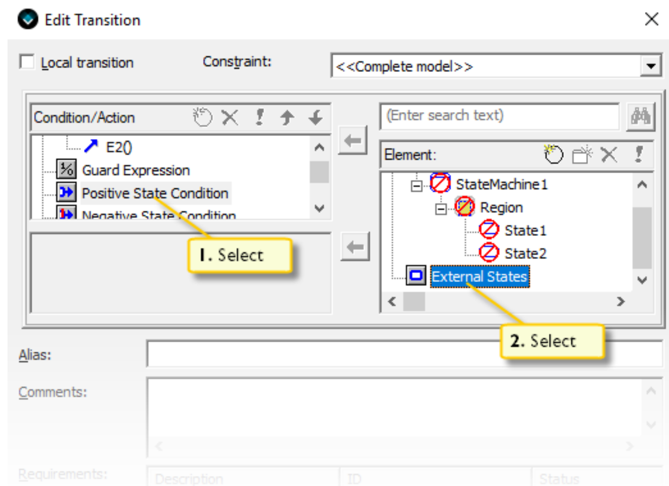
BINDING STATE CONDITIONS

If you want your state machine template to contain a transition that has state conditions—in other words, the transition should be synchronized with another part of the model—you must first create that state condition in the template and then bind the condition with a real state condition in your model. See also *State conditions*, page 170.

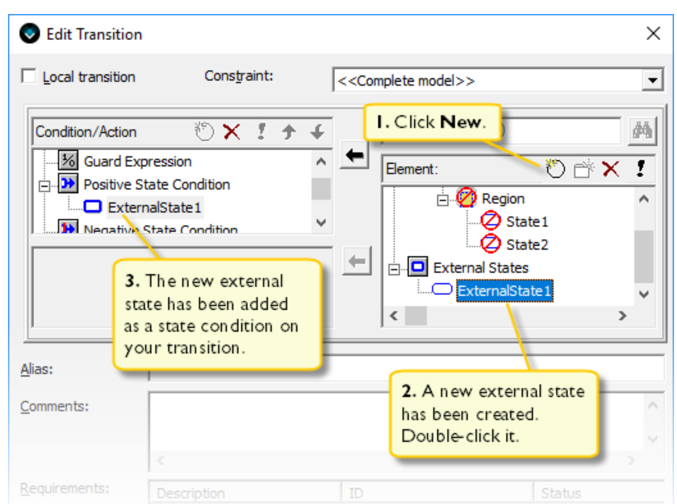
1 In your state machine template, right-click the transition that you want to have a state condition and choose **Edit Transition** from the context menu.

2 Select **Positive State Condition** (or **Negative State Condition** in the left pane, depending on whether the condition should be positive or negative).

If the state condition you want to bind with is in the template, select Internal States. If the state condition is outside the template, select the **External States** symbol in the right pane (as in this example):



3 Add a new bindable external state to your state condition by clicking the **New** icon in the toolbar above the right pane.



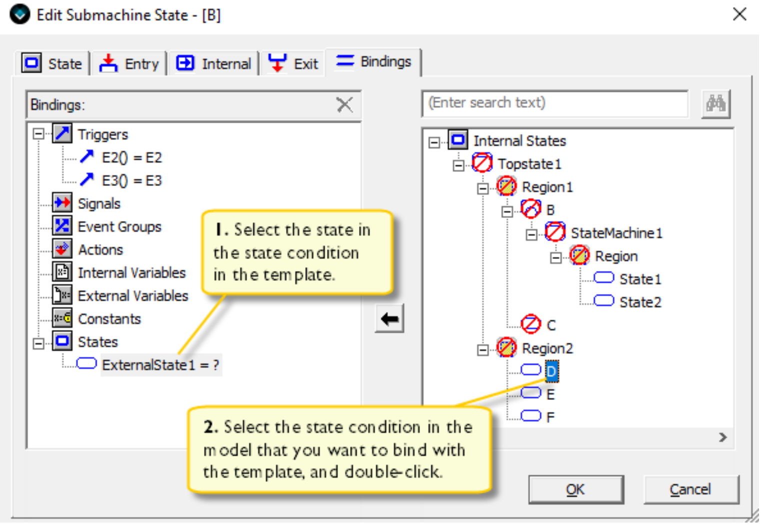
Double-click the external state that you just created to add it as a state condition on your transition.

Consider renaming the new external states to give them more suitable names. If you have already added other external states in the state machine template, they will be displayed here.

Inside a state machine template you cannot refer to states outside the template itself, unless you use the external states as described here and bind the actual state to refer to, when you set up bindings in the submachine state.

4 Now it is time to bind the state condition in your template with a state condition in your state machine model.

In your state machine diagram, select the submachine state (in this example `B`). In the **Associate with template** field, specify the relevant template.



Click the **Bindings** tab and select the state for the state condition in the template (`ExternalState`), then click the state in the model (`D`) that you want to bind with the state condition in the template.

The `D` state condition is now bound with the state condition in the template.

Using variants and features

- Introduction to variants and features
- Working with variants and features

Introduction to variants and features

Learn more about:

- *Variants*, page 217
- *Features*, page 217
- *Include/exclude parts in a variant*, page 218

VARIANTS

If your product will be available for the end user in multiple similar variants, you can define *variants* in the Designer, and mark up parts of your model as belonging to one of these variants, so that they are included in the resulting code, and in testing. This way you can avoid having to maintain two or more separate software development tracks.

The resulting model will contain the states, transitions, and transition elements that match the setup for the variant in the Designer. When you generate code, you choose one of the designed variants, or the complete model.

Note: Using variants is optional. All existing models will work as previously designed.

FEATURES

The Designer also supports defining product functionality (subsets of the design) that can optionally be part of the model as *features*. Each feature has a type that determines how you can include/exclude it in a variant meant for code generation. For information about feature types, see *Edit Features dialog box*, page 253.

You can use the variants tool without using features. The features functionality is only needed if you have parts of the model that should be included in more than one of your runtime variants.

Features are arranged in a *feature tree* that shows the type of the features in the model and their relationship with other features.

INCLUDE/EXCLUDE PARTS IN A VARIANT

The mechanic to set up which parts of the model to include in a variant or in a feature is called *constraints*. A constraint restricts a part of the model to either one of the variants, or to one of the features in the model. When the runtime model is generated, all items that do not have constraints or are constrained to the variant are included, together with all items that are constrained to a feature that is part of the variant.

Constraint is inherited; all states or regions inside a constrained state or region are also constrained. Note that an explicitly constrained state below a state/region that is constrained will still be constrained even if the state/region above is changed to have no constraint.

Note that a transition is excluded from the runtime model if it has:

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

Transitions with 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.

Working with variants and features

What do you want to do?

- *Defining a new feature in your model*, page 218
- *Defining a new variant in your model*, page 219
- *Including a region in a variant or feature*, page 220
- *Including a transition in a variant or feature*, page 220
- *Including a state in a variant or feature*, page 221
- *Including a transition element in a variant or feature*, page 221

DEFINING A NEW FEATURE IN YOUR MODEL

These steps describe how you create a new root feature for your model. By repeating these steps using the **New Sibling** and **New Child** buttons, you can populate the feature tree with more features.

1 In the Designer, open your project.

2 Choose **Edit>Edit Features**. In the **Edit Features** dialog box that is displayed, click the **New Child Feature** button ( ) on the **Action** toolbar, to create a new root feature.

3 Type a name for the feature in the **Name** text box, that describes the part of the model that you will constrain to this feature, like `Dimmer` or `Subwoofer`.

4 Choose which type you want the feature to have, using the **Type** dropdown menu:

- A **Mandatory** feature must be used in all variants if its parent is included.
- An **Optional** feature may be used in a variant if its parent is included.
- If an **Alternative** feature has one or more siblings, you must use exactly one of the siblings in a variant if their parent is included.
- If an **Or** feature has one or more siblings, you must use one or more of the siblings of the Or type in a variant if their parent is included.

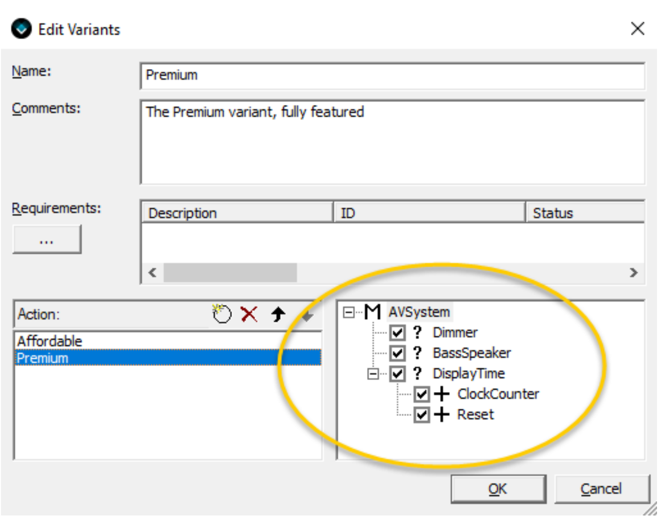
DEFINING A NEW VARIANT IN YOUR MODEL

1 In the Designer, open your project.

2 Choose **Edit>Edit Variants**. In the **Edit Variants** dialog box that is displayed, click the **New** button ( ) on the **Action** toolbar, to create a new variant.

3 Type a name for the variant in the **Name** text box, that describes the product flavor that this variant will be used for, like `Premium` or `Europe`.

4 Specify the features that should be included in the new variant by selecting them in the feature tree on the right-hand side of the dialog box:

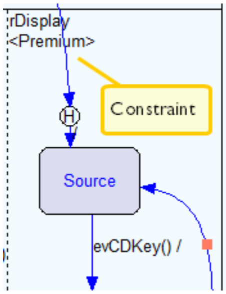


INCLUDING A REGION IN A VARIANT OR FEATURE

If you have regions with states that you want to include in some product variants but exclude from other variants, you can specify constraints for them. The constraints will be inherited by everything below the region in the hierarchic model.

1. In the Designer, open your project.
2. In the state machine diagram, right-click on the region for which you want to specify a constraint, and choose **Edit Region** from the context menu. The **Edit Region** dialog box is displayed.
3. Choose a constraint from the **Constraint** dropdown menu. The menu contains all defined variants and features in your model. See *Include/exclude parts in a variant*, page 218, for a description of the effects of the constraint on the inclusion/exclusion of the region from the runtime variants of your model.

   A region that has been constrained to something else than the complete model, is displayed in the state machine diagram with the name of the constraint below the name of the region:

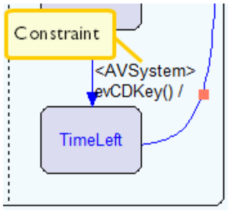


INCLUDING A TRANSITION IN A VARIANT OR FEATURE

If you have transitions that you want to include in some product variants but exclude from other variants, you can specify constraints for them.

1. In the Designer, open your project.
2. In the state machine diagram, right-click on the transition for which you want to specify a constraint, and choose **Edit Transition** from the context menu. The **Edit Transition** dialog box is displayed.
3. Choose a constraint from the **Constraint** dropdown menu. The menu contains all defined variants and features in your model. See *Include/exclude parts in a variant*, page 218, for a description of the effects of the constraint on the inclusion/exclusion of the transition from the runtime variants of your model.

A transition that has been constrained to something else than the complete model, is displayed in the state machine diagram with the name of the constraint above the name of the transition:



INCLUDING A STATE IN A VARIANT OR FEATURE

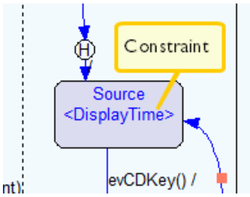
If you have states that you want to include in some product variants but exclude from other variants, you can specify constraints for them.

1 In the Designer, open your project.

2 In the state machine diagram, right-click on the state for which you want to specify a constraint, and choose **Edit State** from the context menu. The **Edit State** dialog box is displayed.

3 Choose a constraint from the **Constraint** dropdown menu. The menu contains all defined variants and features in your model. See *Include/exclude parts in a variant*, page 218, for a description of the effects of the constraint on the inclusion/exclusion of the state from the runtime variants of your model.

A state that has been constrained to something else than the complete model, is displayed in the state machine diagram with the name of the constraint below the name of the state:



INCLUDING A TRANSITION ELEMENT IN A VARIANT OR FEATURE

If you have transition elements that you want to include in some product variants but exclude from other variants, you can specify constraints for them.

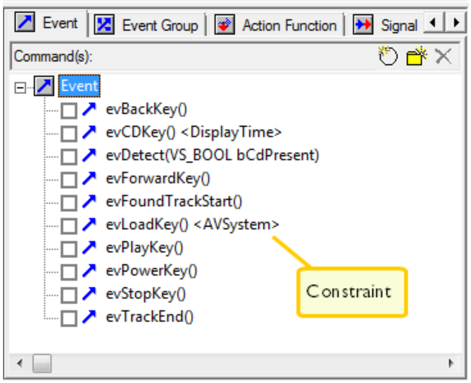
1 In the Designer, open your project.

2 Choose **View>Transition Elements** to open the **Transition Elements** window.

3 Select the transition element for which you want to specify a constraint from the list of elements in the **Commands** pane.

4 Choose a constraint from the **Constraint** dropdown menu in the right-hand pane. The menu contains all defined variants and features in your model. See *Include/exclude parts in a variant*, page 218, for a description of the effects of the constraint on the inclusion/exclusion of the transition element from the runtime variants of your model.

A transition element that has been constrained to something else than the complete model, is displayed in the list of elements with the name of the constraint after the name of the element:



Using requirements files

- Introduction to requirements files
- Working with requirements

Introduction to requirements files

Often, designing state machines is guided by requirements. Formal requirements can be organized using a requirements authoring tool, and can normally be exported in a standardized format called ReqIF (Requirements Interchange Format), an XML file format that can be used to exchange requirements, along with its associated metadata, between software tools from different vendors. Visual State supports integrating ReqIF, currently up to and including version 1.2.

The Visual State Designer can import ReqIF files, and tie objects in your Visual State designs to corresponding requirements, to keep track of how your design fulfills all or some of the requirements.

Working with requirements

What do you want to do?

- *Importing requirements*, page 223
- *Customizing the appearance of requirements in use*, page 224
- *Tying a requirement to a state*, page 224
- *Tying a requirement to a transition*, page 224
- *Tying a requirement to an entry/exit/internal reaction*, page 225
- *Tying a requirement to a transition element*, page 225

IMPORTING REQUIREMENTS

These steps describe how you import requirements from an existing `.reqif` file into the Designer.

1. In the Designer, open the project that you want to make the requirements available in.
2. Choose **File>Import Requirements** and browse to the `.reqif` file that contains the requirement definitions you need.

3. In the **Import Requirements** dialog box, select which attribute definitions that you want to include. Sample values might be displayed to guide you. You can also include them all, and hide some of them in the Designer later. Click **OK** to confirm your selections.
4. The **Requirements Browser** window is displayed to show the available requirements in the Designer.

CUSTOMIZING THE APPEARANCE OF REQUIREMENTS IN USE

To customize how requirements in use are displayed in the Designer:

1. In the Designer, choose **Tools>Customize Appearance**.
2. In the **Customize Appearance** dialog box, select **Requirements** in the **General** category.
3. Select the option **Background color for requirements in use**, and adjust the color using the RGB sliders.

TYING A REQUIREMENT TO A STATE

To tie one or more requirements to a state, follow these steps:

1. In the Designer, open a project that you have imported requirements into.
2. In the state machine diagram, right-click on a state that you want to tie a requirement to, and choose **Edit State** from the context menu. The **Edit State** dialog box is displayed.
3. Click the browse button below the label **Requirements**.
4. In the dialog box **Select Requirements**, use the checkboxes to select the requirements that apply to this state. Then click **OK** to close this dialog box, and **OK** to close the **Edit State** dialog box.

TYING A REQUIREMENT TO A TRANSITION

To tie one or more requirements to a transition, follow these steps:

1. In the Designer, open a project that you have imported requirements into.
2. In the state machine diagram, right-click on a transition that you want to tie a requirement to, and choose **Edit Transition** from the context menu. The **Edit Transition** dialog box is displayed.
3. Click the browse button below the label **Requirements**.

4. In the dialog box **Select Requirements**, use the checkboxes to select the requirements that apply to this state. Then click **OK** to close this dialog box, and **OK** to close the **Edit Transition** dialog box.

TYING A REQUIREMENT TO AN ENTRY/EXIT/INTERNAL REACTION

To tie one or more requirements to an entry/exit/internal reaction, follow these steps:

1. In the Designer, open a project that you have imported requirements into.
2. In the state machine diagram, right-click on a state that contains the reaction that you want to tie a requirement to, and choose **Edit State** from the context menu. The **Edit State** dialog box is displayed.
3. Click on the relevant tab (**Entry**, **Exit**, or **Internal**), and select the reaction in the list of reactions.
4. Click the browse button below the label **Requirements**.
5. In the dialog box **Select Requirements**, use the checkboxes to select the requirements that apply to this reaction. Then click **OK** to close this dialog box, and **OK** to close the **Edit State** dialog box.

TYING A REQUIREMENT TO A TRANSITION ELEMENT

To tie one or more requirements to a transition element, follow these steps:

1. In the Designer, open a project that you have imported requirements into.
2. Choose **View>Transition Elements** to open the **Transition Elements** window.
3. Click on the tab for the relevant transition element type, and then select the element from the list of elements on the **Commands** pane.
4. Click the browse button below the label **Requirements** on the pane to the right.
5. In the dialog box **Select Requirements**, use the checkboxes to select the requirements that apply to this element. Then click **OK** to close this dialog box, and **OK** to close the **Transition Elements** window.

The Visual State Designer

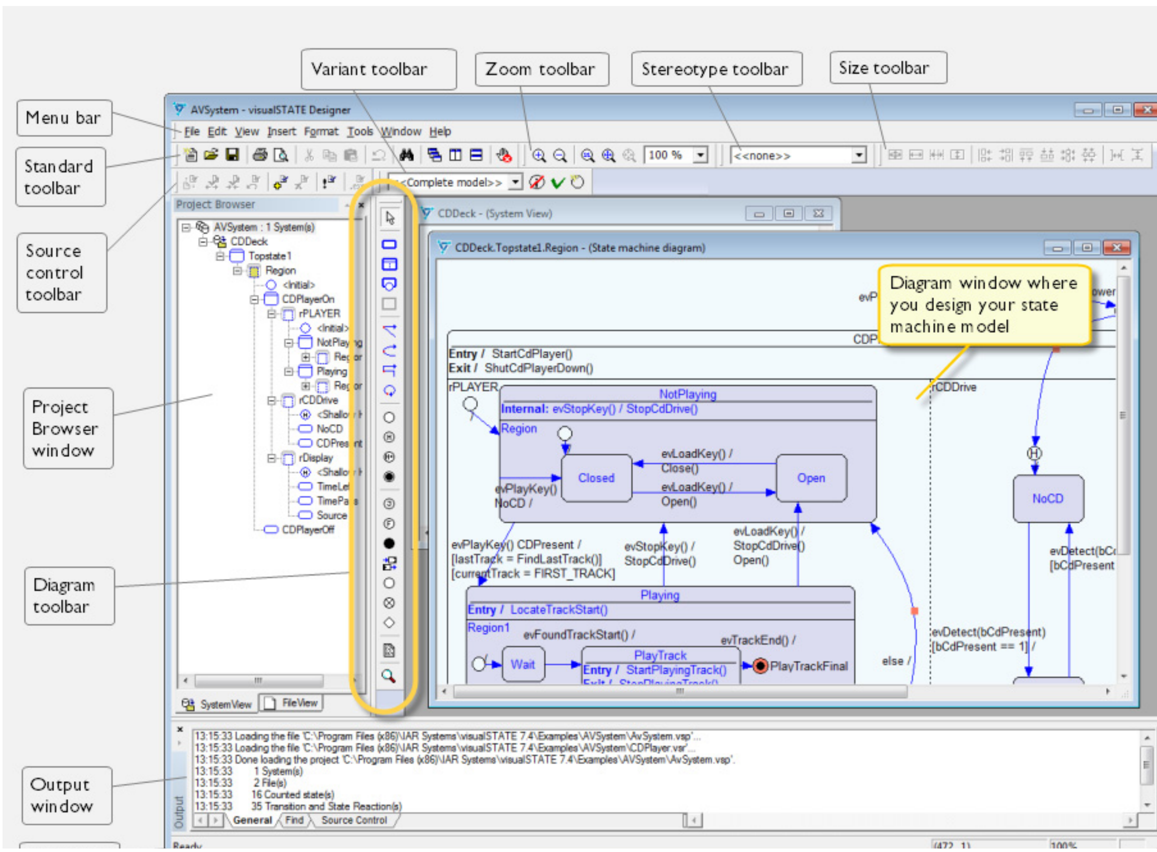
Introduction to the Visual State Designer

Learn more about:

- *Briefly about the Visual State Designer*, page 228

BRIEFLY ABOUT THE VISUAL STATE DESIGNER

The Visual State Designer is a graphical tool for designing state machines by drawing state machine diagrams using the UML notation.



The project created in the Designer can later be imported into a Navigator workspace.

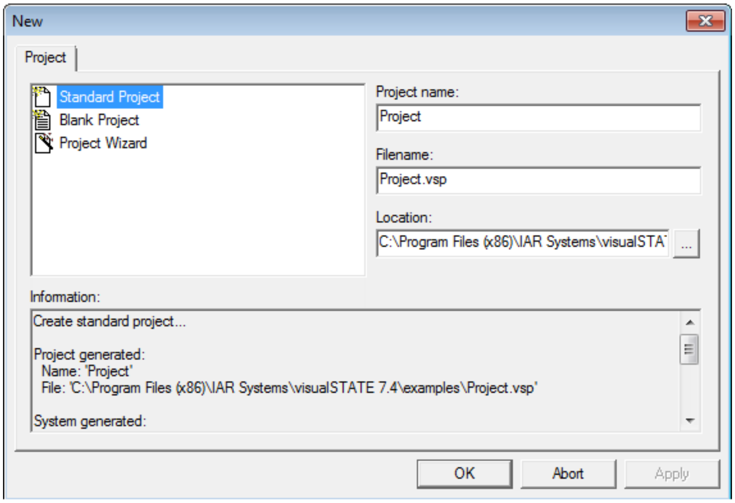
Using the Visual State Designer

What do you want to do?

- *Creating and saving a project with systems and state machine diagrams*, page 229
- *Creating systems and state machine diagrams in a blank project*, page 230
- *Editing objects in the state machine diagram*, page 231

CREATING AND SAVING A PROJECT WITH SYSTEMS AND STATE MACHINE DIAGRAMS

1 In the Designer, choose **File>New**, or click the **New** button ( ) on the standard toolbar.



2 Choose one of the following alternatives:

- **Standard Project**, to create a standard project with one system that contains one top-level state machine. See *Creating a standard workspace*, page 76
- **Blank Project**, to create an empty project where you can create your systems and top-level state machines. See *Creating systems and state machine diagrams in a blank project*, page 230.
- **Project Wizard**, to guide you trough the process of creating a customized project where you can specify the number of systems, top-level state machines, etc.

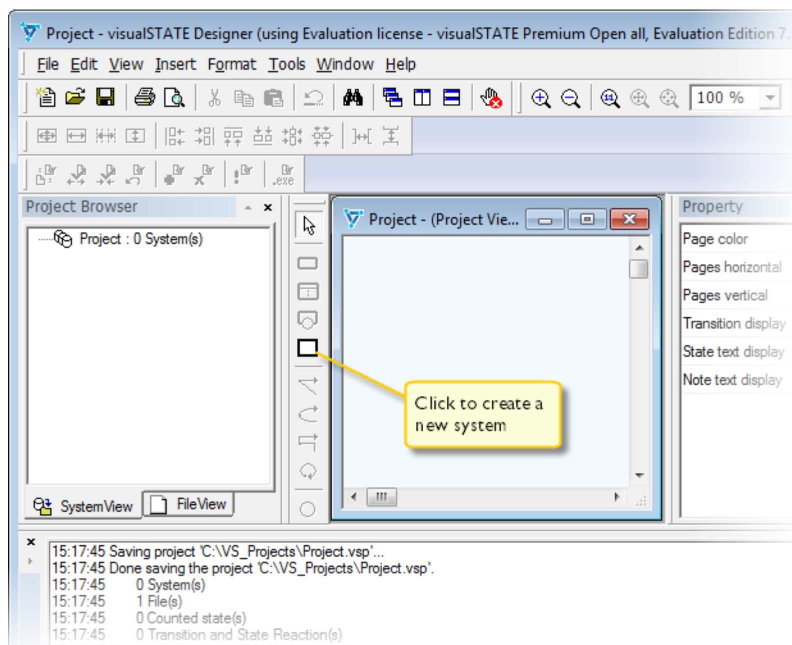
3 Click **OK**.

4 When you have created the project, choose **File>Save Project**.

CREATING SYSTEMS AND STATE MACHINE DIAGRAMS IN A BLANK PROJECT

If you already have created a blank project, you must also create a system and a top-level state machine before you can start designing your state machine model.

1 To create a system, click the **Systems** button (□) on the **Diagram** toolbar.



2 Click in the **Project View** window. A square will be inserted which represents the new system, and the **Project Browser** window will be updated with the new system.

Right-click to deactivate the system tool.

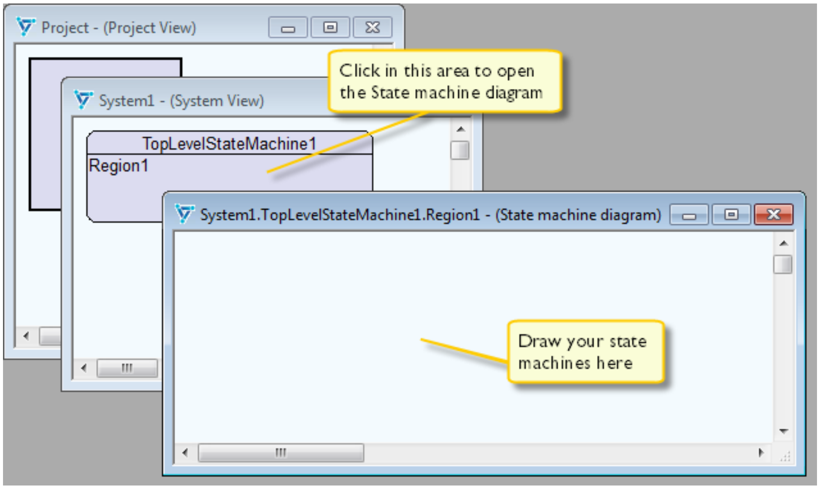
3 Double-click in the new system (but not on the system name). The **System View** window is displayed.

4 On the **Diagram** toolbar, click the **Composite State** button (⊟).

5 Click in the **System View** window. A top-level state machine will be inserted and represents a new state machine diagram file which will contain your state machine model. The **Project Browser** window will be updated.

Right-click to deactivate the tool.

6 Double-click in the new top-level state machine (but not on its name). The **State machine diagram** window is displayed:



7 You can now start designing your state machine model with states and transitions, see *Designing state machines*, page 126.

8 On the **Designer** menu, choose **File>Save Project**.

EDITING OBJECTS IN THE STATE MACHINE DIAGRAM

In the Designer, you can rename and change descriptions for, for example projects, systems, top-level state machines, regions, states, composite states.

1 In the state machine diagram, click the objects to edit.

2 To rename an object, type a new name and press Enter, or select the item in the **Project Browser** window and choose **Rename** from the context menu.

3 To enter or change alias names and description notes, select an object in the state machine diagram and choose **Edit** from the context menu. An edit dialog box is displayed where you can enter and edit aliases and descriptions.

For renaming of transition elements, see *Working with transition elements and transition element files*, page 183.

INSERTING NOTES

1 On the **Diagram** toolbar, click the **Note** button ().

2 In the state machine diagram, click where you want to place the note. A rectangle is inserted.

3 Right-click to deactivate the Note tool.

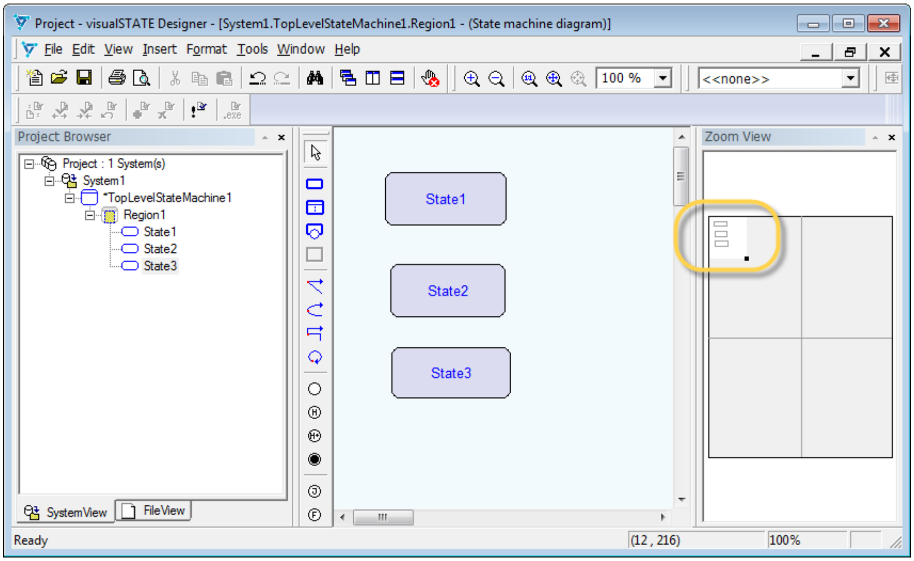
4 To write text in the note, click the frame—it will be marked with blank squares—and start typing. To insert a line break, press Ctrl+Enter.

5 Press Enter to finish typing.

6 Optional: To select an image to show in the note, right-click in the note and choose **Edit Note** from the context menu. Then use the **Browse** button in the dialog box to navigate to an image to add.

NAVIGATING IN THE STATE MACHINE DIAGRAM

1 Choose **View>Zoom View**. In the **Zoom View** window, the gray area represents the entire state machine diagram and the white square represents the current view in the **Diagram View** window.



2 To view a different part of the state machine, drag the white square to a different location. If you use the scroll bars of the **Diagram View** window, the movement is reflected also in the **Zoom View** window.

To get detailed information about an object, just point at it with the mouse pointer in the **Diagram View** window.

GETTING WARNINGS FOR NON-VERIFIABLE ELEMENTS

If during design you want to receive a warning when you create or use a non-verifiable element, you can use *safe mode*.

1 To set safe-mode, click the **Safe Mode** button ( ) on the **Standard** toolbar or choose **Tools>Safe Mode**.

2 A warning will be given during the design process if a non-verifiable element is used.

For information about non-verifiable elements, see *Non-verifiable elements*, page 417.

IMPORTING C HEADER FILES INTO THE PROJECT OR TOP-LEVEL STATE MACHINE

You can reuse an existing C header file that contains function declarations and constants for your Visual State project.

The function declarations and constants must have a special syntax for the Designer to recognize them.

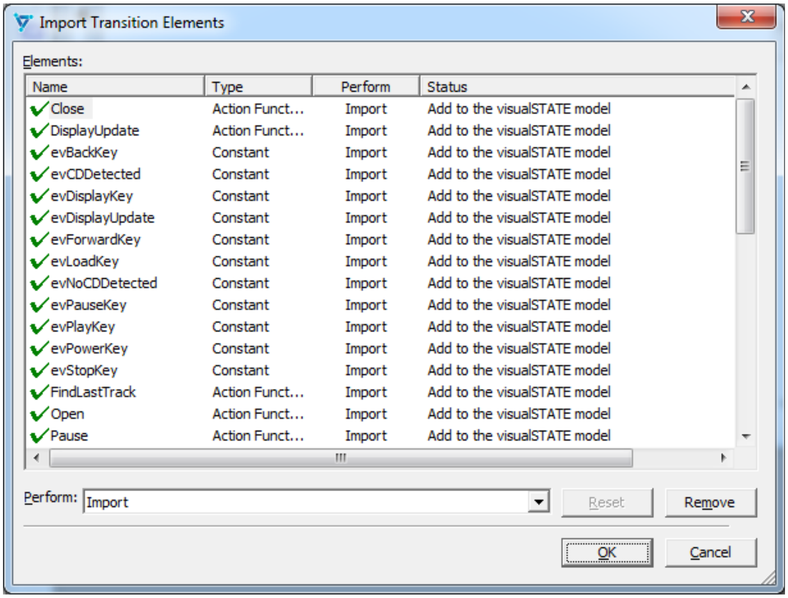
The function declarations in the imported header file map to action functions in IAR Visual State, whereas constants in the header file map to constants in IAR Visual State.

1 In the Designer, open your project.

2 In the **Project Browser** window, click the **File View** tab. Select the project or top-level state machine to import to, right-click and choose **Import** from the context menu.

3 In the **Import** dialog box, specify the file to import and click **OK**.

4 The header files will be loaded and analyzed for function declarations and constants. The **Import Transition Elements** dialog box is displayed.



5 Select the items to import and click **OK**. The selected items will be imported and displayed in the **Transition Element** window.

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

See also *Syntax of C header files*, page 316.

CREATING MULTIPLE SYSTEM INSTANCES

You can create multiple instances of a system. Typically, this is useful if you want to control multiple identical hardware or software units by means of the same state machine design.

1. In the Designer, open your project.
2. In the **Project Browser** window, select the system for which you want to specify multiple instances and choose **Edit** from the context menu.
3. In the **Edit System** dialog box, specify the number of instances you want to create.

See also *The Visual State system*, page 123.

USING DESIGNER BACKUP FILES

1. Open a Windows Explorer window.
2. Locate the `vssm`, `vsp`, and `vsr` backup files you want to use, for example `Name1.vssm.bk1`, `Name2.vsp.bk1`, and `Name3.vsr.bk1`.

   Note: The files must have the same `bk` extension number, for example `vsp.bk1` and `vsr.bk1`.
3. In the file browser, remove the `bk#` extensions.
4. In the Designer, open the `vsp` file you have renamed.

   Your project will be loaded with the backed up `vssm`, `vsp`, and `vsr` files.

To change the backup settings, see *Settings dialog box*, page 289.

CUSTOMIZING THE DESIGNER

The Designer can be customized with regard to handling of files, states, transitions, message display, etc.

1. In the Designer, choose **Tools>Settings** to open the **Settings** dialog box.

   To customize the graphical appearance, choose **Tools>Customize Appearance**.
2. Click the appropriate tab and make your settings. See *Settings dialog box*, page 289 and *Customize Appearance dialog box*, page 242, respectively.

   The settings are stored in the registry (colors and fonts are stored in the project).

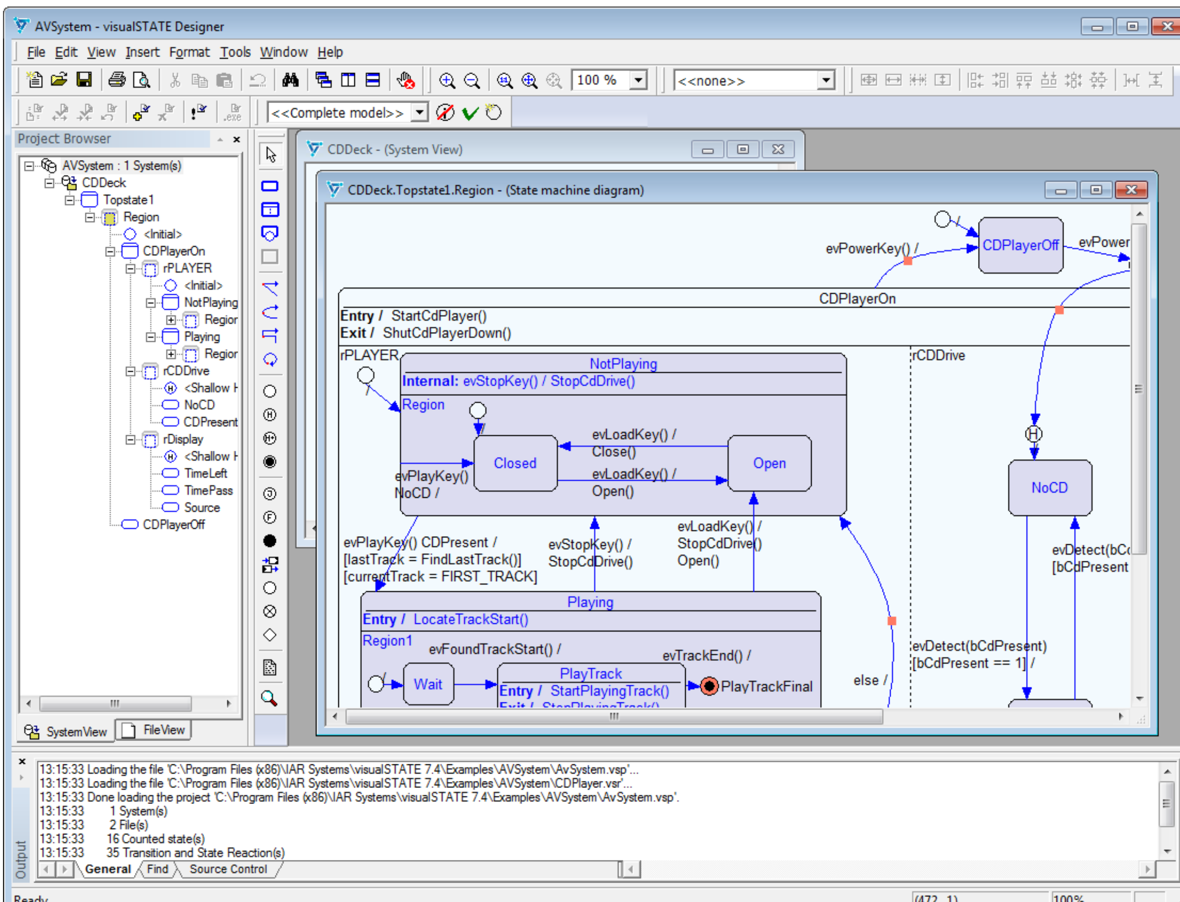
Graphical environment for the Designer

Reference information about:

- *Transition Elements window*, page 295
- *Zoom View window*, page 298
- *General Designer windows context menus*, page 298

The Designer main window

The main window is displayed when you start the Designer.



The main window of the Designer is a container for displaying the **Project Browser** window, the diagram windows, and the **Output** window.

Menu bar

The menu bar contains:

File

Commands for creating, opening, and saving projects, importing functions, declarations, and requirements, printing, and exiting the Designer.

Edit

Standard Windows commands for working with text.

View

Commands for opening windows, toggling grid assistance, zooming, and controlling which toolbars to display.

Insert

Commands for inserting systems, states, and transitions, and notes.

Format

Commands for adjusting the size, alignment, and position of objects.

Tools

Commands for using the selection and the zoom tools, making settings for grid assistance, toggling safe mode, customizing the appearance of the objects, and making general Designer settings.

Window

Commands for changing how the Designer windows are arranged on the screen.

Help

Commands that provide help about the Designer.

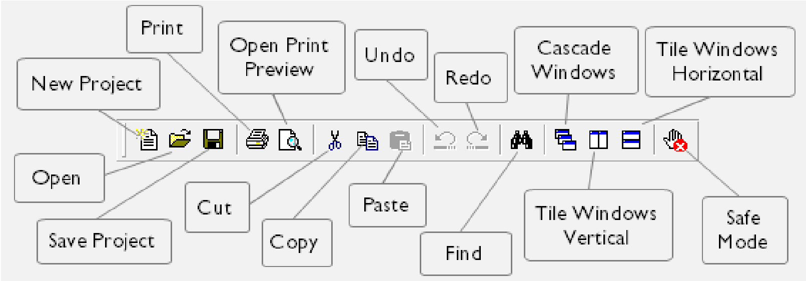
See also *Reference information on Designer menus*, page 302.

Standard toolbar

The **Standard** toolbar—available from the **View** menu—provides buttons for the most useful commands on the Designer menus.

For a description of a button, point to it with the mouse pointer. The name of the button is displayed as a tooltip and a description is displayed in the status bar at the bottom of the main window. When a command is not available, the corresponding toolbar button is dimmed, and you will not be able to click it.

This figure shows the menu commands that correspond to each of the toolbar buttons:



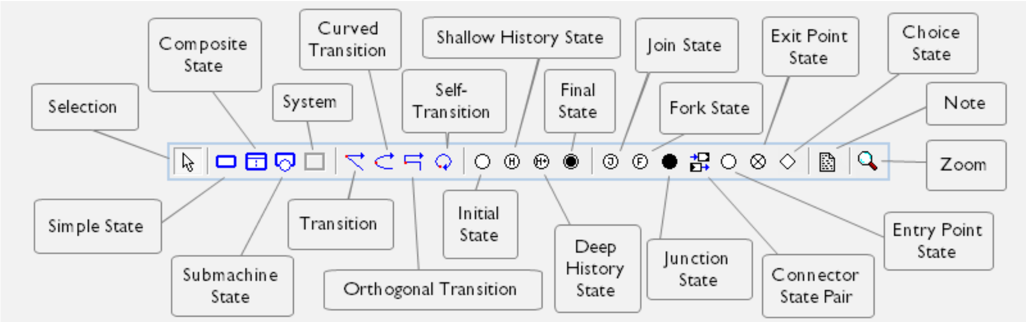
See also *File menu*, page 303 and *Edit menu*, page 305.

Diagram toolbar

The **Diagram** toolbar—available from the **View** menu—provides buttons for drawing.

For a description of a button, point to it with the mouse pointer. The name of the button is displayed as a tooltip and a description is displayed in the status bar at the bottom of the main window. When a command is not available, the corresponding toolbar button is dimmed, and you will not be able to click it.

This figure shows the menu commands that correspond to each of the toolbar buttons:



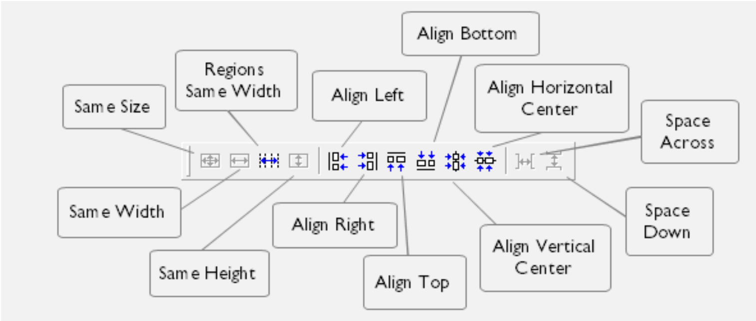
See also *Insert menu*, page 308.

Size toolbar

The **Size** toolbar—available from the **View** menu—provides buttons for positioning and modifying selected objects in the **Project View**, **System View**, and **View** windows.

For a description of a button, point to it with the mouse pointer. The name of the button is displayed as a tooltip and a description is displayed in the status bar at the bottom of the main window. When a command is not available, the corresponding toolbar button is dimmed, and you will not be able to click it.

This figure shows the menu commands that correspond to each of the toolbar buttons:



See also *Format menu*, page 310.

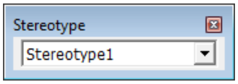
Stereotype toolbar

The **Stereotype** toolbar—available from the **View** menu—provides a means of creating states with a uniform look.

When the **Stereotype** toolbar has an active stereotype, any new state that you create will inherit properties and behavior from the active stereotype.

To activate another stereotype, choose it from the **Stereotype** toolbar drop-down menu. To revert to creating new states that are not based on a stereotype, activate the **<<none>>** stereotype.

This figure shows the **Stereotype** toolbar:



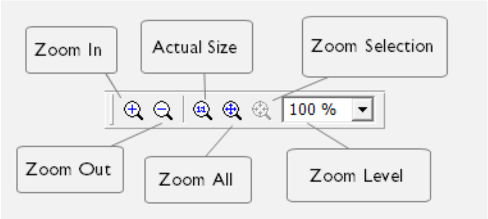
See also *Stereotypes for creating states with a uniform look*, page 140.

Zoom toolbar

The **Zoom** toolbar—available from the **View** menu—provides buttons for zooming in the **Project View**, **System View**, and **Digram View** windows.

For a description of a button, point to it with the mouse pointer. The name of the button is displayed as a tooltip and a description is displayed in the status bar at the bottom of the main window. When a command is not available, the corresponding toolbar button is dimmed, and you will not be able to click it.

This figure shows the menu commands that correspond to each of the toolbar buttons:

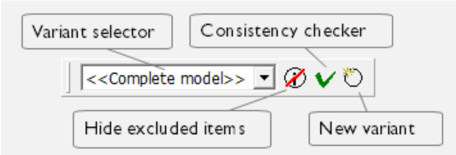


See also *Tools menu*, page 311.

Variant toolbar

The Variant toolbar—available from the **View** menu—provides buttons for working with product variants in the model.

This figure shows the toolbar:



Variant selector

Choose which product *variant* to be active in the Designer. If you choose **<<Complete model>>**, the entire model will be active.

Hide excluded items

Hides all transition elements in the model that are not used by the selected variant.

Consistency checker

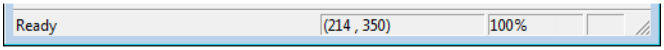
Performs a quick consistency check of the model before you open the model in the other Visual State components, restricting the model to the selected active variant. Errors are listed in the **Output** window.

New variant

Displays the **Edit Variants** dialog box, where you can define a new variant; see *Edit Variants dialog box*, page 274.

Status bar

The status bar at the bottom of the window can be enabled from the **View** menu.

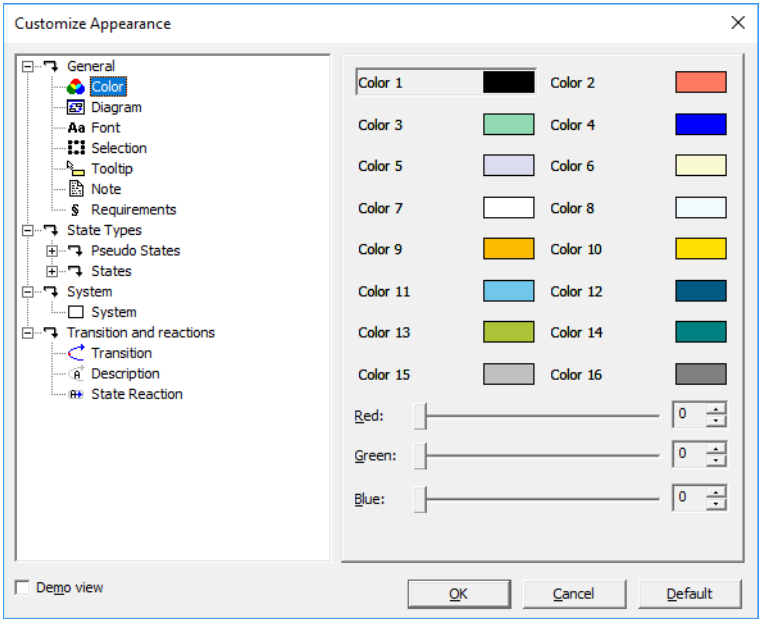


The status bar displays:

- Descriptions of menu commands when you open a menu and hover over commands
- Descriptions of toolbar buttons that you point to
- Mouse positions and zoom scale.

Customize Appearance dialog box

The **Customize Appearance** dialog box is available from the **Tools** menu.



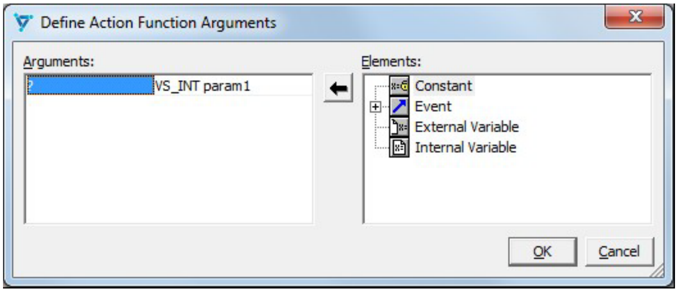
Use this dialog box to specify the appearance of graphical elements, for example fonts, colors, etc.

Demo view

Displays a preview of your settings.

Define Action Function Arguments dialog box

The **Define Action Function Arguments** dialog box is available from the **Edit State** dialog box and the **Edit Transition** dialog box:



Use this dialog box to define arguments for action functions.

See *Specifying arguments for action function parameters*, page 185.

Arguments

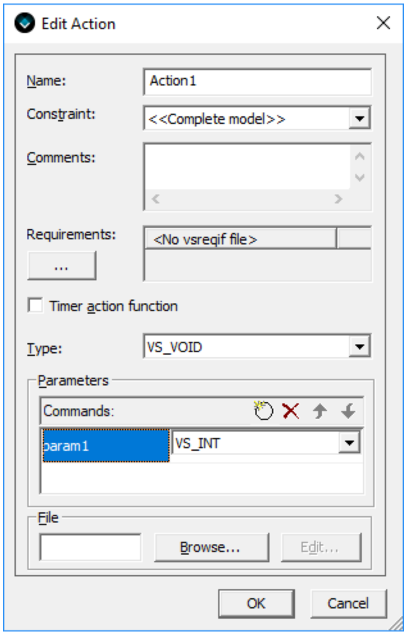
Displays the arguments for the function.

Elements

Displays the transition elements that you can use as arguments. Double-click to choose one.

Edit Action dialog box

The **Edit Action** dialog box is available from the **Edit State** and **Edit Transition** dialog boxes.



Use this dialog box to create an action function.

See *Declaring action functions in external C files*, page 192.

Name

Specify a name for the action function.

Constraint

Choose which parts of the model that the action function will be available in.

Comments

Type a description for the action function.

Requirements

Shows which formal requirements that are tied to this item. Click the browse button to open the **Select Requirements** window where you can edit which formal requirements to tie to this item; see *Select Requirements window*, page 288.

Timer action function

Changes the action function to a timer action function that takes two parameters. Any related changes will be done too. For example, changing the number of arguments to function calls for the action function.

Type

Select the return type of the action function.

Parameters

To create or edit the action function parameters, there are four buttons available:

New

Creates a new parameter.

Delete

Deletes the selected parameter in the list.

Move Up

Moves the selected parameter upward in the list.

Move Down

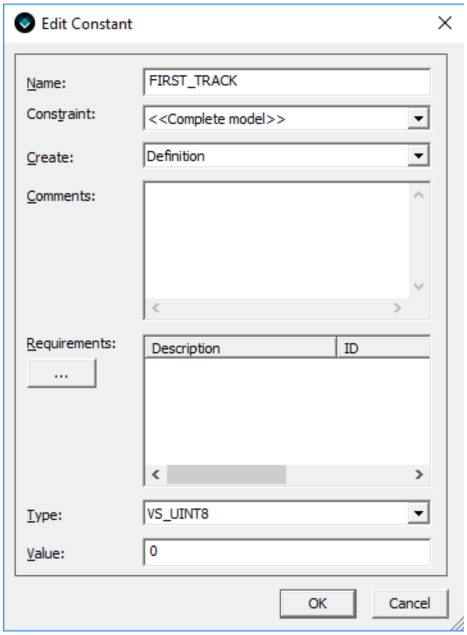
Moves the selected parameter downward in the list.

File

If there are action functions declared and implemented in an external C file, click **Browse** to locate and load it. To edit the file, click **Edit**. The external editor that you have specified on the **Tools>Settings>External editor** page will be opened. Edit the file and save when you are finished.

Edit Constant dialog box

The **Edit Constant** dialog box is available from the **Edit State** and **Edit Transition** dialog boxes.



Use this dialog box to create a constant.

Name

Specify a name for the constant.

Constraint

Choose which parts of the model that the constant will be available in.

Create

Select whether the constant is a declaration or a definition.

Comments

Type a description for the constant.

Requirements

Shows which formal requirements that are tied to this item. Click the browse button to open the **Select Requirements** window where you can edit which formal requirements to tie to this item; see *Select Requirements window*, page 288.

Type

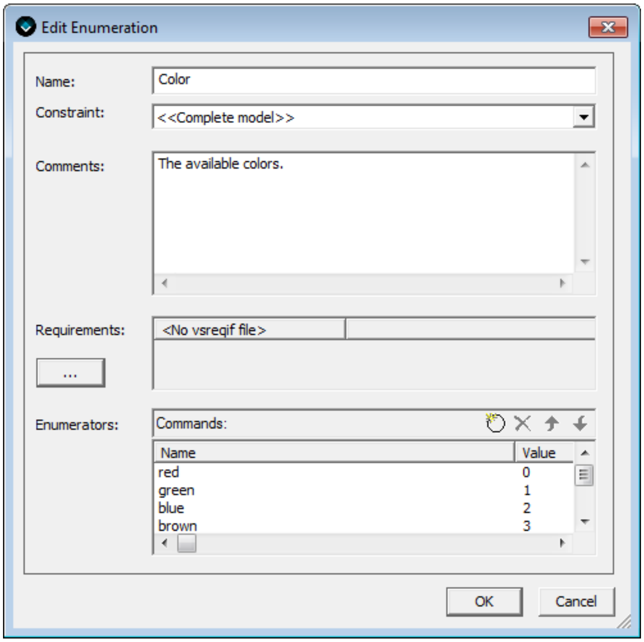
Select the data type of the constant.

Value

Specify the value of the constant.

Edit Enumeration dialog box

The **Edit Enumeration** dialog box is available from the **Edit State** and **Edit Transition** dialog boxes.



Use this dialog box to create an enumeration.

Name

Specify a name for the enumeration.

Constraint

Choose which parts of the model that the enumeration will be available in.

Comments

Type a description for the enumeration.

Requirements

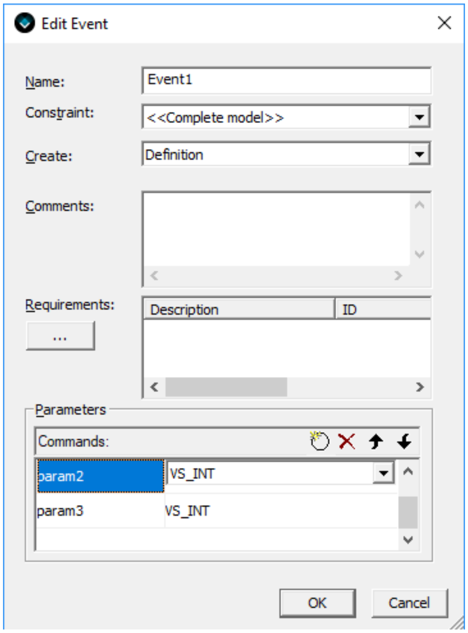
Shows which formal requirements that are tied to this item. Click the browse button to open the **Select Requirements** window where you can edit which formal requirements to tie to this item; see *Select Requirements window*, page 288.

Enumerators

Add enumerators to the enumeration. The names can be edited in the list, but not the values.

Edit Event dialog box

The **Edit Event** dialog box is available from the **Edit State** and **Edit Transition** dialog boxes.



Use this dialog box to create an event.

See *Creating a transition element*, page 184.

Name

Specify a name for the event.

Constraint

Choose which parts of the model that the event will be available in.

Create

Select whether the event is a declaration or a definition.

Comments

Type a description for the event.

Requirements

Shows which formal requirements that are tied to this item. Click the browse button to open the **Select Requirements** window where you can edit which formal requirements to tie to this item; see *Select Requirements window*, page 288.

Parameters

To create or edit the event parameters, there are four buttons available:

New

Creates a new parameter.

Delete

Deletes the selected parameter in the list.

Move Up

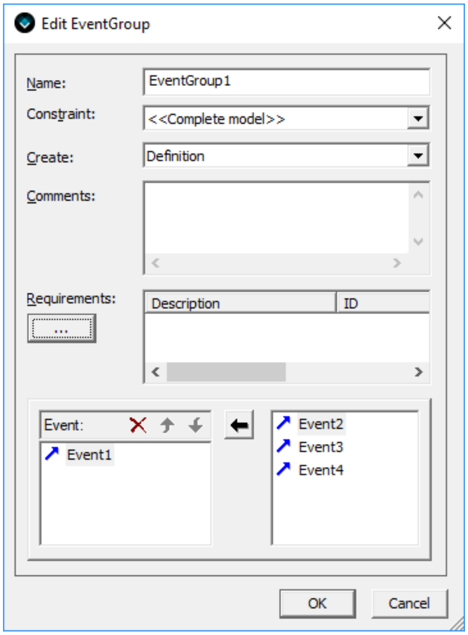
Moves the selected parameter upward in the list.

Move Down

Moves the selected parameter downward in the list.

Edit Event Group dialog box

The **Edit Event Group** dialog box is available from the **Edit State** and **Edit Transition** dialog boxes.



Use this dialog box to create an event group.

Name

Specify a name for the event group.

Constraint

Choose which parts of the model that the event group will be available in.

Create

Select whether the event group is a declaration or a definition.

Comments

Type a description for the event group.

Requirements

Shows which formal requirements that are tied to this item. Click the browse button to open the **Select Requirements** window where you can edit which formal requirements to tie to this item; see *Select Requirements window*, page 288.

Event

Populate the event group with events—that you have previously created—from the list to the right:

Add Event

Adds the selected event to the event group. Double-clicking it will have the same effect.

Remove

Removes the selected event from the event group.

Move Up

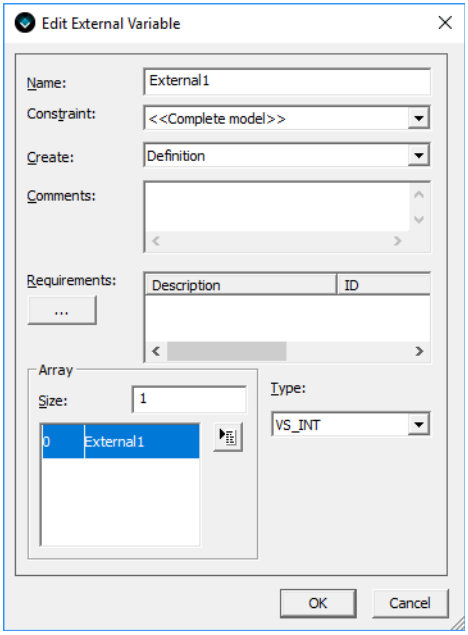
Moves the selected item upward in the list.

Move Down

Moves the selected item downward in the list.

Edit External Variable dialog box

The **Edit External Variable** dialog box is available from the **Edit State** and **Edit Transition** dialog boxes.



Use this dialog box to create an external variable.

See *Creating a transition element*, page 184.

Name

Specify a name for the external variable.

Constraint

Choose which parts of the model that the external variable will be available in.

Create

Select whether the external variable is a declaration or a definition.

Comments

Type a description for the external variable.

Requirements

Shows which formal requirements that are tied to this item. Click the browse button to open the **Select Requirements** window where you can edit which formal requirements to tie to this item; see *Select Requirements window*, page 288.

Type

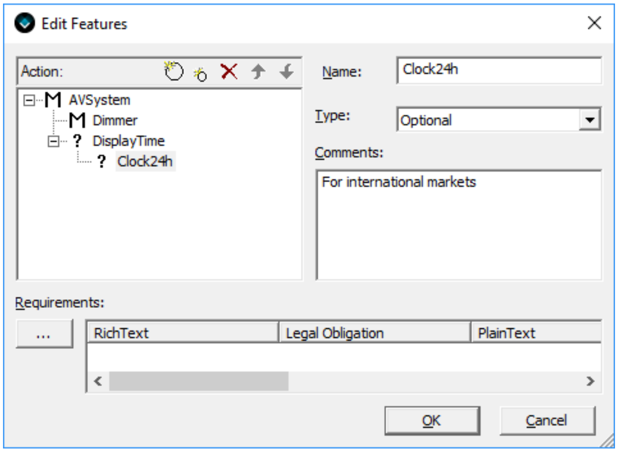
Select the data type of the external variable.

Array

If the variable is an array, specify its **Size** and (if you are defining it), the initial values of the elements in it. To edit the values of all array elements, click the **Set All** button to open the context menu.

Edit Features dialog box

The **Edit Features** dialog box is available from the **Edit** menu.



Use this dialog box to define or edit a feature and to edit the feature tree.

See *Defining a new feature in your model*, page 218.

Action

Displays a list with the features that have been defined.

Select an item in the list and use one of these buttons:

New Sibling

Creates a new sibling to the selected feature.

New Child

Creates a new child to the selected feature.

Delete

Deletes the selected feature in the list, together with any children.

Move Up

Moves the selected feature upward in the list.

Move Down

Moves the selected feature downward in the list.

The feature tree can also be modified by dragging features to new places in the tree.

Name

Specify a name for the feature.

Type

Choose a type for the feature. Choose between:

Mandatory

A Mandatory feature must be used in all variants if its parent is included.

Optional

An Optional feature may be used in a variant if its parent is included.

Alternative

If an Alternative feature has one or more siblings, you must use exactly one of the siblings in a variant if their parent is included.

Or

If an Or feature has one or more siblings, you must use one or more of the siblings of the Or type in a variant if their parent is included.

Comments

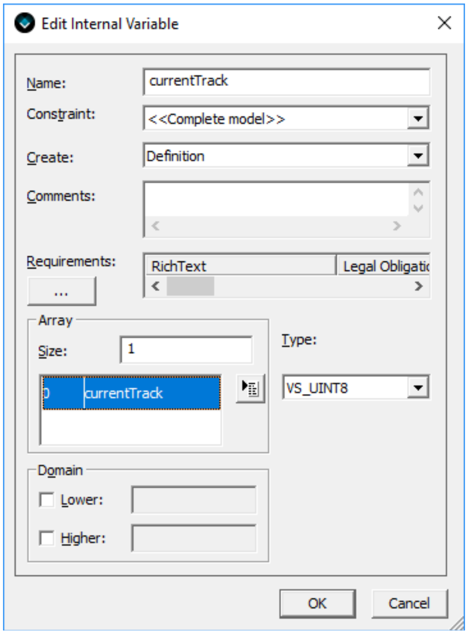
Type a description for the feature.

Requirements

Shows which formal requirements that are tied to this item. Click the browse button to open the **Select Requirements** window where you can edit which formal requirements to tie to this item; see *Select Requirements window*, page 288.

Edit Internal Variable dialog box

The **Edit Internal Variable** dialog box is available from the **Edit State** and **Edit Transition** dialog boxes.



Use this dialog box to create an internal variable.

Name

Specify a name for the internal variable.

Constraint

Choose which parts of the model that the internal variable will be available in.

Create

Select whether the internal variable is a declaration or a definition.

Comments

Type a description for the internal variable.

Requirements

Shows which formal requirements that are tied to this item. Click the browse button to open the **Select Requirements** window where you can edit which formal requirements to tie to this item; see *Select Requirements window*, page 288.

Array

If the variable is an array, specify its **Size** and (if you are defining it), the initial values of the elements in it. To edit the values of all array elements, click the **Set All** button to open the context menu.

Type

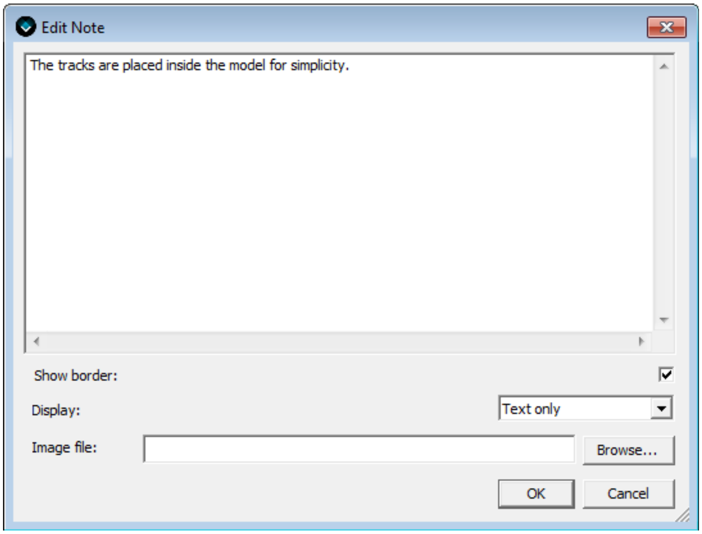
Select the data type of the internal variable.

Domain

If the value of the variable should be within a given range, specify the **Lower** and the **Higher** end of the range. The domain can be checked by the Verificator.

Edit Note dialog box

The **Edit Note** dialog box is available by right-clicking on a note in a Designer view window.



Use this dialog box to edit the text and the properties of a note.

See *Inserting notes*, page 232.

Show border

Displays a frame around the note in the diagram.

Display

Choose which parts of the note to display. Choose between:

Text only

Displays just the text part of the note.

Image only

Displays just the image part of the note. If no image has been specified to display in the note, the note will be empty.

Text and image

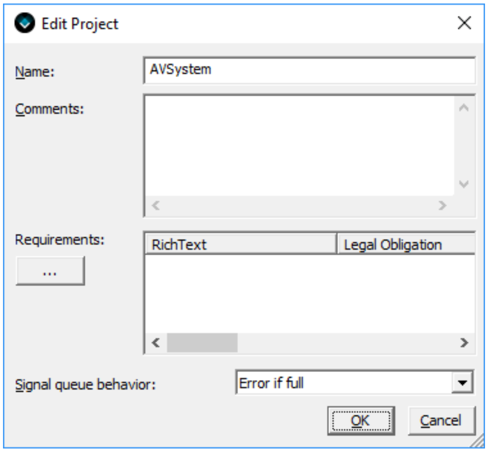
Displays both the text and the image part of the note.

Image file

Use the **Browse** button to select an image to display in the note.

Edit Project dialog box

The **Edit Project** dialog box is available by right-clicking on a project in the **Project Browser** window.



Use this dialog box to specify the properties of a project.

See *Creating a new project in a workspace*, page 77.

Name

Specify a name for the project.

Comments

Type any comments that describe the project. They will be included in the documentation report.

Requirements

Shows which formal requirements that are tied to this item. Click the browse button to open the **Select Requirements** window where you can edit which formal requirements to tie to this item; see *Select Requirements window*, page 288.

Signal queue behavior

Specify what should happen if the signal queue fills up. Choose between:

Drop if Full

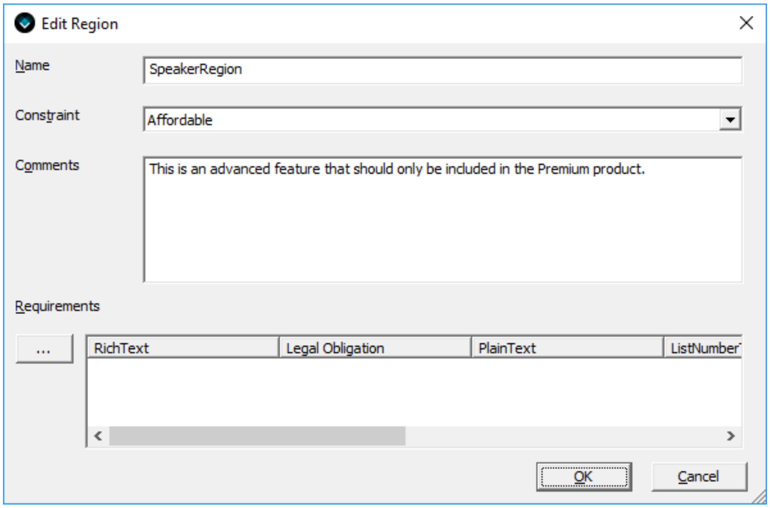
Drops the signal completely in the runtime code if the queue is full when a signal is meant to be inserted.

Error if Full

Returns an error from the deduct function in the runtime code if the queue is full when a signal is meant to be inserted.

Edit Region dialog box

The **Edit Region** dialog box is available by right-clicking on a region in the **State machine diagram** window.



Use this dialog box to set a constraint for a region.

See *Including a region in a variant or feature*, page 220.

Name

Specify a name for the region.

Constraint

Choose which parts of the model that the region will be included in.

Comments

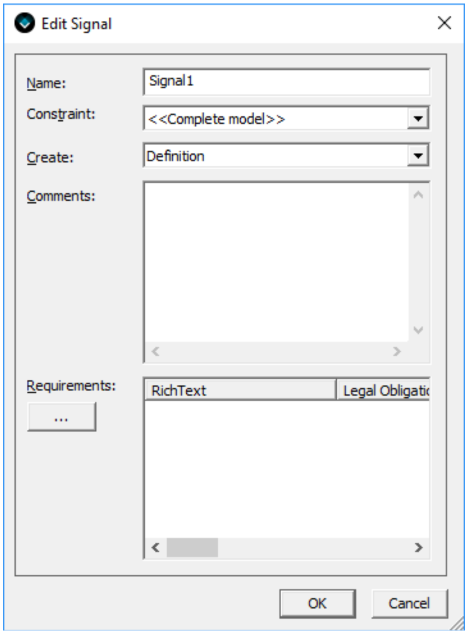
Type a description for the region.

Requirements

Shows which formal requirements that are tied to this item. Click the browse button to open the **Select Requirements** window where you can edit which formal requirements to tie to this item; see *Select Requirements window*, page 288.

Edit Signal dialog box

The **Edit Signal** dialog box is available from the **Edit State** and **Edit Transition** dialog boxes.



Use this dialog box to create a signal.

See *Specifying the signal queue behavior and size*, page 190.

Name

Specify a name for the signal.

Constraint

Choose which parts of the model that the signal will be available in.

Create

Select whether the signal is a declaration or a definition.

Comments

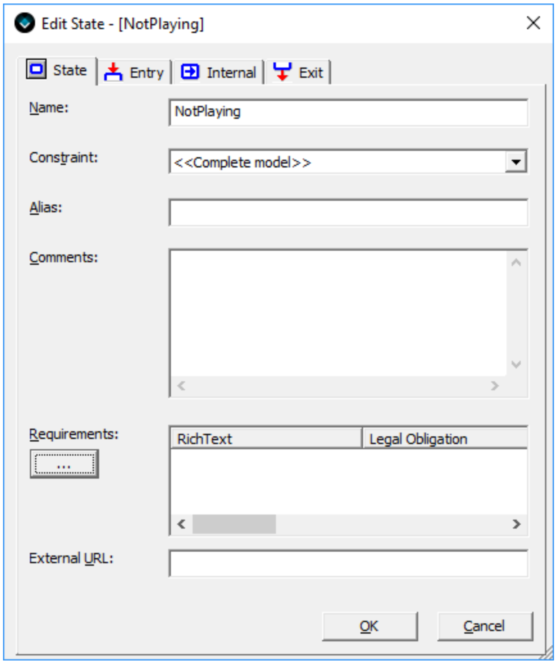
Type a description for the signal.

Requirements

Shows which formal requirements that are tied to this item. Click the browse button to open the **Select Requirements** window where you can edit which formal requirements to tie to this item; see *Select Requirements window*, page 288.

Edit State dialog box

The **Edit State** dialog box is available by right-clicking on a state in the **State machine diagram** window.



Use this dialog box to specify the properties of a state, see *Identifying and drawing simple states*, page 128. The **Entry**, **Internal**, and **Exit** pages contain options for editing the entry, internal, and exit reactions for the state, see *Edit State dialog box : state reactions*, page 263.

Name

Specify a name for the state.

Constraint

Choose which parts of the model that the state will be included in.

Alias

Specify an alias (optional name) for the state. The alias is only used for display and is not used when you generate code.

Comments

Type any comments that describe the state. They will be included in the documentation report.

Requirements

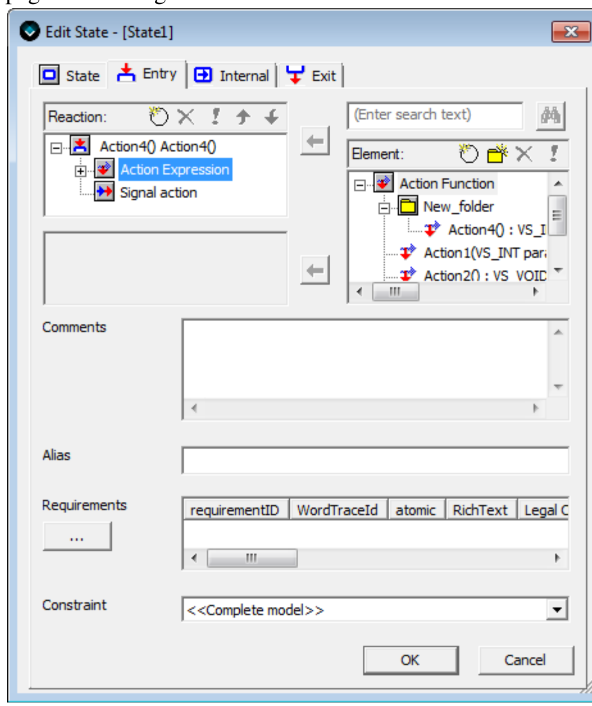
Shows which formal requirements that are tied to this item. Click the browse button to open the **Select Requirements** window where you can edit which formal requirements to tie to this item; see *Select Requirements window*, page 288.

External URL

Enter a URL, for instance `http://www.iar.com`, to create a link in the top right corner of the state's name compartment. Clicking the link when the state is selected opens the URL in your web browser.

Edit State dialog box : state reactions

The **Edit State** dialog box—see *Edit State dialog box*, page 261—has three tabbed pages for editing state reactions.



Use this the options on this tab page to edit the entry, internal, and exit reactions for the state. See also *State reactions*, page 150.

Reaction

To manage the state reactions, there are five buttons available. Select an item in the list and use one of these buttons:

New

Inserts a new state reaction or transition element based on the selected type.

Delete

Deletes the selected item in the list.

Define

Opens the **Define Action Function Parameters** dialog box, see *Define Action Function Arguments dialog box*, page 243.

Move Up

Moves the selected item upward in the list.

Move Down

Moves the selected item downward in the list.

Text field just under the Reactions area

Use this text field for editing guard expressions and action expressions. See *Adding assignments and guard expressions*, page 187.

Find

Specify text to search for in the transition elements and click the button.

Element

Displays a list with the transition elements that have been defined.

Select an item in the list and use one of these buttons:

Apply Element

Applies the selected transition element to the state reaction. Double-clicking it will have the same effect.

New

Opens a dialog box where you can create a new transition element.

New Folder

Creates a new folder where you can collect your transition elements.

Delete

Deletes the selected item in the list.

Define

Opens a dialog box where you can edit the selected item.

Comments

Type any comments that describe the reaction. They will be included in the documentation report.

Alias

Specify an alias (optional name) for the reaction. The alias is only used for display and is not used when you generate code.

Requirements

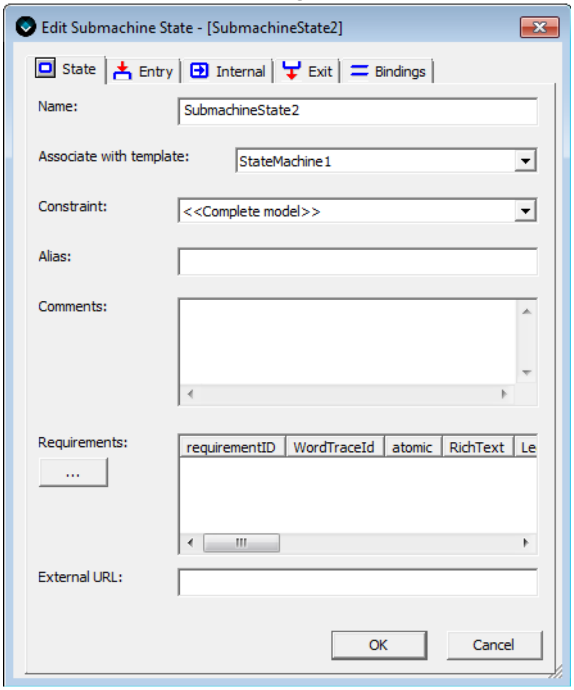
Shows which formal requirements that are tied to this item. Click the browse button to open the **Select Requirements** window where you can edit which formal requirements to tie to this item; see *Select Requirements window*, page 288.

Constraint

Choose which parts of the model that the reaction will be included in.

Edit Submachine State dialog box

The **Edit Submachine State** dialog box is available by right-clicking on a submachine state in the **State machine diagram** window.



Use this dialog box to specify the properties of a submachine state, thus instantiating a state machine template, see *Reusing designs using state machine templates*, page 201 and *Instantiating a state machine template*, page 207.

The **Entry**, **Internal**, and **Exit** pages contain options for editing the entry, internal, and exit reactions for the submachine state, see *Edit State dialog box : state reactions*, page 263, and the page **Bindings** contains options for binding transition elements in submachine state with transition elements in the state machine template, see *Edit Submachine State dialog box : bindings*, page 269.

Name

Specify a name for the submachine state.

Associate with template

Chooses which state machine template the submachine state should be associated with.

Constraint

Choose which parts of the model that the submachine state will be included in.

Alias

Specify an alias (optional name) for the submachine state. The alias is only used for display and is not used when you generate code.

Comments

Type any comments that describe the submachine state. They will be included in the documentation report.

Requirements

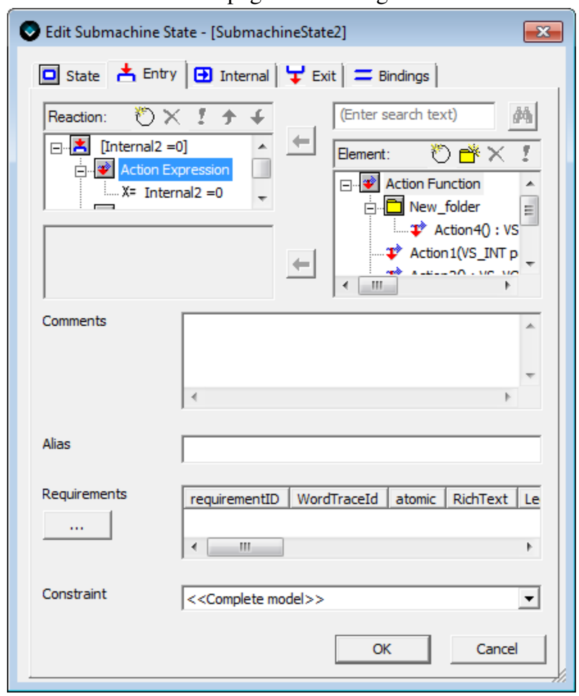
Shows which formal requirements that are tied to this item. Click the browse button to open the **Select Requirements** window where you can edit which formal requirements to tie to this item; see *Select Requirements window*, page 288.

External URL

Enter a URL, for instance `http://www.iar.com`, to create a link in the top right corner of the submachine state's name compartment. Clicking the link when the submachine state is selected opens the URL in your web browser.

Edit Submachine State dialog box : state reactions

The **Edit Submachine State** dialog box—see *Edit Submachine State dialog box*, page 265—has three tabbed pages for editing state reactions.



Use this the options on this tab page to edit the entry, internal, and exit reactions for the submachine state. See also *State reactions*, page 150.

Reaction

To manage the state reactions, there are five buttons available. Select an item in the list and use one of these buttons:

New

Inserts a new state reaction or transition element based on the selected type.

Delete

Deletes the selected item in the list.

Define

Opens the **Define Action Function Parameters** dialog box, see *Define Action Function Arguments dialog box*, page 243.

Move Up

Moves the selected item upward in the list.

Move Down

Moves the selected item downward in the list.

Text field just under the Reactions area

Use this text field for editing guard expressions and action expressions. See *Adding assignments and guard expressions*, page 187.

Find

Specify text to search for in the transition elements and click the button.

Element

Displays a list with the transition elements that have been defined.

Select an item in the list and use one of these buttons:

Apply Element

Applies the selected transition element to the state reaction. Double-clicking it will have the same effect.

New

Opens a dialog box where you can create a new transition element.

New Folder

Creates a new folder where you can collect your transition elements.

Delete

Deletes the selected item in the list.

Define

Opens a dialog box where you can edit the selected item.

Comments

Type any comments that describe the reaction. They will be included in the documentation report.

Alias

Specify an alias (optional name) for the reaction. The alias is only used for display and is not used when you generate code.

Requirements

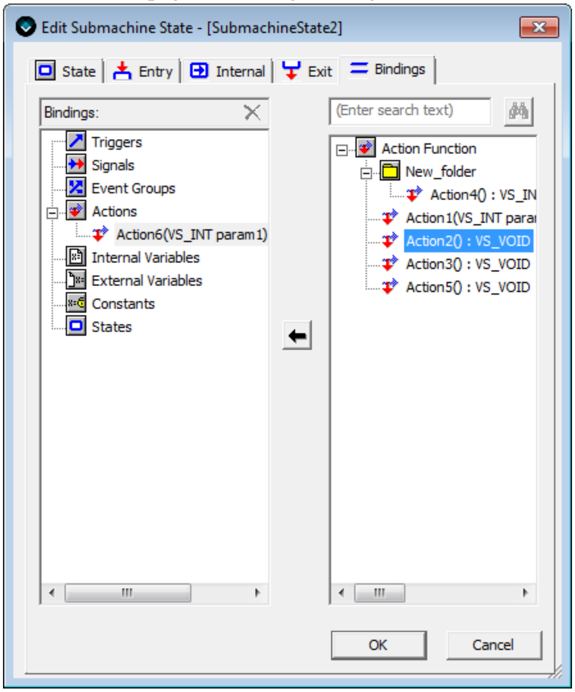
Shows which formal requirements that are tied to this item. Click the browse button to open the **Select Requirements** window where you can edit which formal requirements to tie to this item; see *Select Requirements window*, page 288.

Constraint

Choose which parts of the model that the reaction will be included in.

Edit Submachine State dialog box : bindings

The **Edit Submachine State** dialog box—see *Edit Submachine State dialog box*, page 265—a tabbed page for editing bindings.



Use this the options on this tab page to bind transition elements in the submachine state with abstract transition elements in the state machine template.

Bindings

Select a reaction in the list to display the transition elements that have been defined in the state machine model. Choose an element from the list to the right to bind it with a transition element in the state machine template.

Note that when you bind state actions, you must specify whether the state action is internal in the template or external outside the template.

Apply

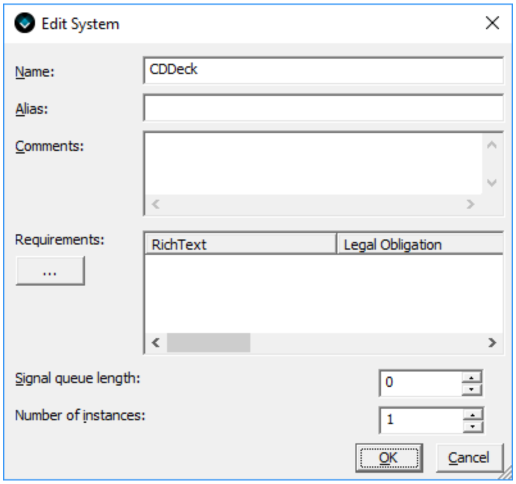
Applies the selected transition element to the state reaction. Double-clicking it will have the same effect.

Find

Specify text to search for in the list of transition elements and click the button.

Edit System dialog box

The **Edit System** dialog box is available by right-clicking on a system in a Designer view window.



Use this dialog box to specify the properties of a system.

See *The Visual State system*, page 123

Name

Specify a name for the system.

Alias

Specify an alias (optional name) for the system. The alias is only used for display and is not used when you generate code.

Comments

Type any comments that describe the system. They will be included in the documentation report.

Requirements

Shows which formal requirements that are tied to this item. Click the browse button to open the **Select Requirements** window where you can edit which formal requirements to tie to this item; see *Select Requirements window*, page 288.

Signal queue length

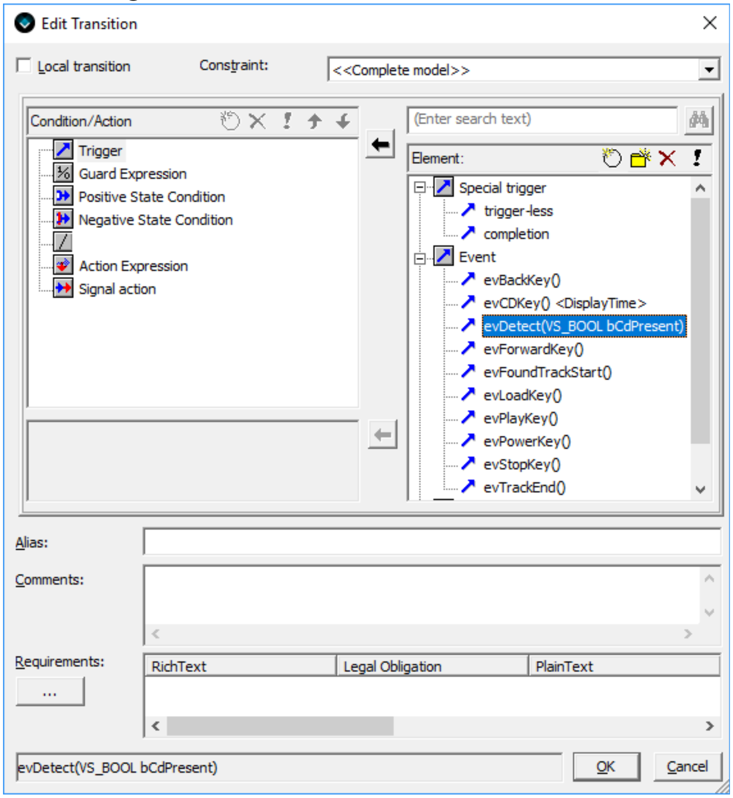
Specify the length of the signal queue.

Number of instances

Specify the number of instances of the system.

Edit Transition dialog box

The **Edit Transition** dialog box is available by right-clicking on a transition in the **State machine diagram** window.



Use this dialog box to create a transition condition and action.

See also *Creating transitions*, page 175.

Local transition

Specify whether the transition should be *local*, see *Local transitions*, page 173.

Constraint

Choose which parts of the model that the transition will be included in, see *Include/exclude parts in a variant*, page 218.

Find

Specify text to search for in the transition elements and click the button.

Condition/Action

Displays the transition elements that you have defined for your condition and action.

To create or edit the transition conditions and actions, there are five buttons available. Select an item in the list and use one of these buttons:

New

Inserts a new rule based on the selected type.

Delete

Deletes the selected item in the list.

Define

Displays the **Define Action Function Parameters** dialog box, see *Define Action Function Arguments dialog box*, page 243.

Move Up

Moves the selected item upward in the list.

Move Down

Moves the selected item downward in the list.

Text field just under the Condition/Action area

Use this text field for editing guard expressions and action expressions. See *Adding assignments and guard expressions*, page 187.

Element

Displays a list with the elements that have been defined for the selected transition. For information about transition elements, see *Introduction to transition elements*, page 177.

Select an item in the list and use one of these buttons:

Apply Element

Applies the selected element to the transition. Double-clicking it will have the same effect.

New

Opens a dialog box where you can create a new transition element.

New Folder

Creates a new folder where you can collect your transition elements.

Delete

Deletes the selected item in the list.

Define

Opens a dialog box where you can edit the selected item.

Alias

Specify an alias (optional name) for the transition. The alias is only used for display and is not used when you generate code.

Comments

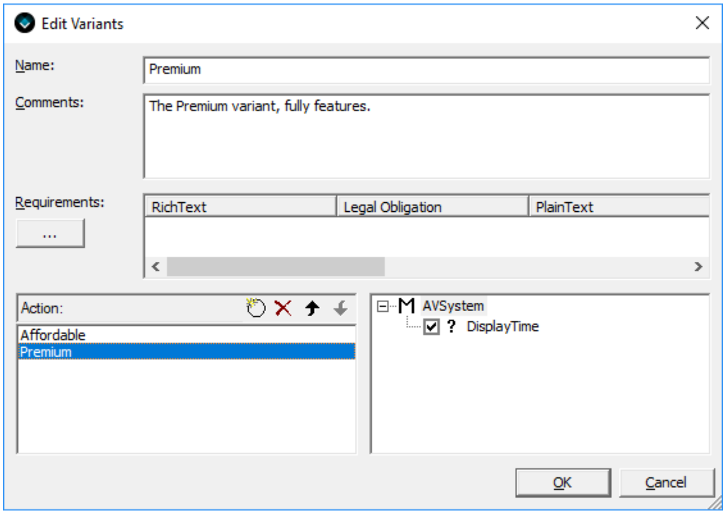
Type any comments that describe the transition. They will be included in the documentation report.

Requirements

Shows which formal requirements that are tied to this item. Click the browse button to open the **Select Requirements** window where you can edit which formal requirements to tie to this item; see *Select Requirements window*, page 288.

Edit Variants dialog box

The **Edit Variants** dialog box is available from the **Edit** menu.



Use this dialog box to define or edit a variant.

See *Defining a new variant in your model*, page 219.

Name

Specify a name for the variant.

Comments

Type a description for the variant.

Requirements

Shows which formal requirements that are tied to this item. Click the browse button to open the **Select Requirements** window where you can edit which formal requirements to tie to this item; see *Select Requirements window*, page 288.

Action

Displays the variants that you have defined for your product.

To create or edit the variants, there are four buttons available. Select an item in the list and use one of these buttons:

New

Creates a new variant.

Delete

Deletes the selected variant in the list.

Move Up

Moves the selected variant upward in the list.

Move Down

Moves the selected variant downward in the list.

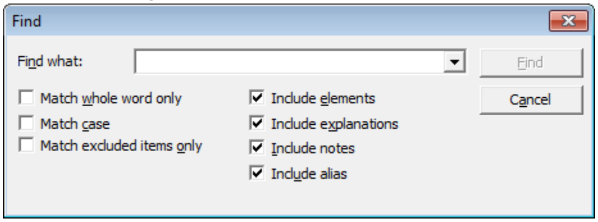
Features

Displays the tree of features that have been defined for the model.

Include/exclude features in the selected variant by selecting/deselecting the checkboxes. Which features that can be included/excluded is determined by the type of the feature, see *Edit Features dialog box*, page 253.

Find dialog box

The **Find** dialog box is available from the **Edit** menu.



Use this dialog box to find text and transition elements in projects, systems, and top-level state machines.

See *Searching for a transition element*, page 193.

Match whole word only

Searches for the specified text only if it occurs as a separate word. Otherwise, specifying `int` will also find `print`, `sprintf` etc.

Match case

Searches only for occurrences that exactly match the case of the specified text. Otherwise, specifying `int` will also find `INT` and `Int`.

Match excluded items only

Searches for the specified text only in objects that have been excluded from your project.

Include elements

Searches for the specified text in transition elements.

Include explanations

Searches for the specified text in comments.

Include notes

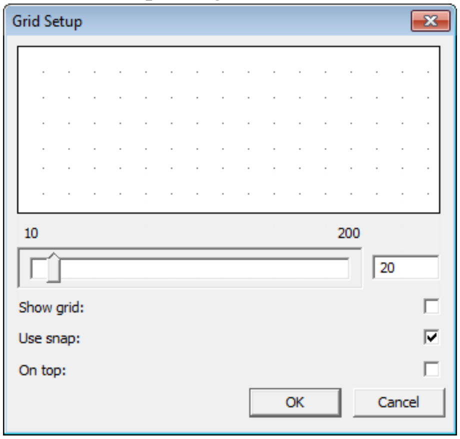
Searches for the specified text in notes.

Include alias

Searches for the specified text in aliases.

Grid Setup dialog box

The **Grid Setup** dialog box is available from the **Tools** menu.



Use this dialog box to set up the graphical support grid.

Slider

Sets the density of the grid lines to a value from 10 to 200 pixels.

Show grid

Displays the grid.

Use snap

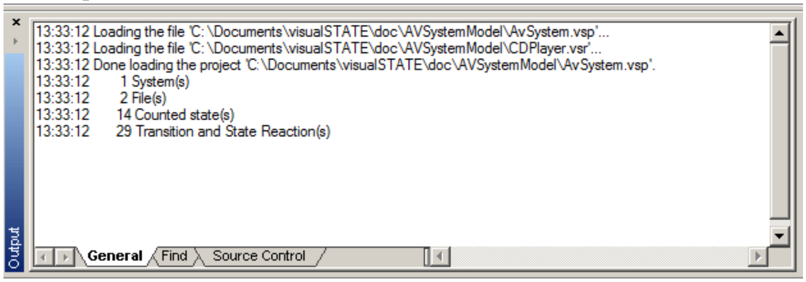
Makes graphical objects in diagrams snap to the grid when you move them (including when the grid is invisible).

On top

Displays the grid on top of all objects in the diagrams.

Output window

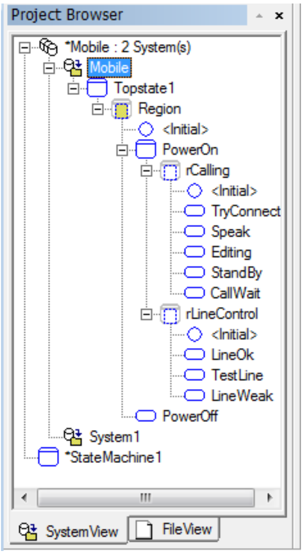
The **Output** window is available from the **View** menu.



This window displays information about the loaded project, results of reaction element searches, and undo actions.

Project Browser window

The **Project Browser** window is available from the **View** menu.



This window is a browser where you can see the structure of the loaded Visual State project.

See *Creating and saving a project with systems and state machine diagrams*, page 229

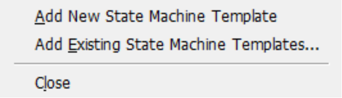
The window has two different views:

- **File View**—which shows the file structure of the project, with systems and state machine templates.
- **System View**—which shows the model structure of the Visual State project in detail. This view shows systems and individual states, and state machine templates.

To go to a system, state, or state machine template in a Designer view window, double-click it in the **Project Browser** window.

General context menu

This context menu is available by right-clicking on the background in the **Project Browser** window:



These commands are available:

Add New State Machine Template

Adds a new state machine template. See also *Reusing designs using state machine templates*, page 201.

Add Existing State Machine Templates

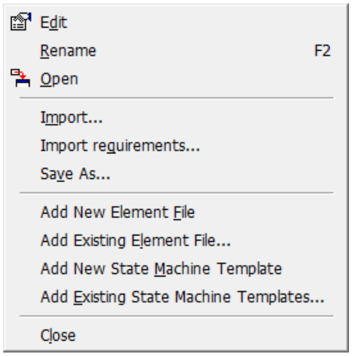
Displays a dialog box where you can locate existing state machine templates.

Close

Closes the window.

Project context menu

This context menu is available by right-clicking on a project file in the **Project Browser** window:



These commands are available:

Edit

Displays the **Edit Project** dialog box, see *Edit Project dialog box*, page 257.

Rename

Selects the name of the file so you can edit it.

Open

Opens a **Project View** window for the system, see *Project View window*, page 284.

Import

Opens the **Import Transition Elements** dialog box, see *Importing C header files into the project or top-level state machine*, page 234.

Import Requirements

Opens the **Import Requirements** dialog box, see *Importing requirements*, page 223.

Save As

Displays a standard dialog box where you can save the selected file under a new name.

Add New Element File

Creates and adds a new transition element file to the project, see *Creating and adding a new transition element file*, page 193.

Add Existing Element File

Adds an existing transition element file to the project, see *Adding an existing transition element file*, page 193.

Add New State Machine Template

Adds a new state machine template. See also *Reusing designs using state machine templates*, page 201.

Add Existing State Machine Templates

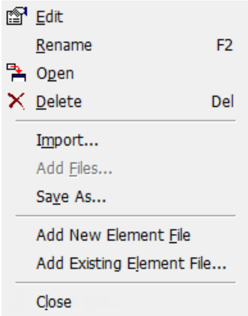
Displays a dialog box where you can locate existing state machine templates.

Close

Closes the window.

Top-level state machine context menu

This context menu is available by right-clicking on a top-level state machine file in the **Project Browser** window:



These commands are available:

Edit

Displays the **Edit State** dialog box, see *Edit State dialog box*, page 261.

Rename

Selects the name of the file so you can edit it.

Open

Opens a diagram window containing the top-level state machine.

Delete

Deletes the selected file.

Import

Opens the **Import Transition Elements** dialog box, see *Importing C header files into the project or top-level state machine*, page 234.

Add Files

Displays a standard dialog box where you can locate existing files to add to the system.

Save As

Displays a standard dialog box where you can save the selected file under a new name.

Add New Element File

Creates and adds a new transition element file to the project, see *Creating and adding a new transition element file*, page 193.

Add Existing Element File

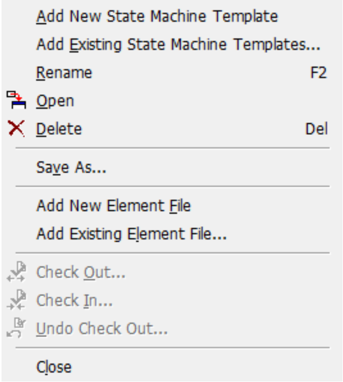
Adds an existing transition element file to the project, see *Adding an existing transition element file*, page 193.

Close

Closes the window.

State machine template context menu

This context menu is available by right-clicking on a state machine template file in the **Project Browser** window:



These commands are available:

Add New State Machine Template

Adds a new state machine template. See also *Reusing designs using state machine templates*, page 201.

Add Existing State Machine Templates

Displays a dialog box where you can locate existing state machine templates.

Rename

Selects the name of the file so you can edit it.

Open

Opens a diagram window containing the state machine template.

Delete

Deletes the selected file.

Save As

Displays a standard dialog box where you can save the selected file under a new name.

Add New Element File

Creates and adds a new transition element file to the project, see *Creating and adding a new transition element file*, page 193.

Add Existing Element File

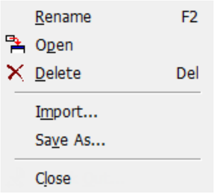
Adds an existing transition element file to the project, see *Adding an existing transition element file*, page 193.

Close

Closes the window.

Transition element context menu

This context menu is available by right-clicking on a transition element file in the **Project Browser** window:



These commands are available:

Rename

Selects the name of the file so you can edit it.

Open

Opens the **Transition Elements** window, see *Transition Elements window*, page 295.

Delete

Deletes the selected file.

Import

Opens the **Import Transition Elements** dialog box, see *Importing C header files into the project or top-level state machine*, page 234.

Save As

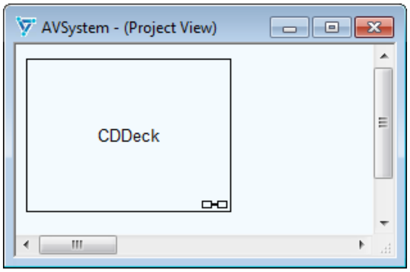
Displays a standard dialog box where you can save the selected file under a new name.

Close

Closes the window.

Project View window

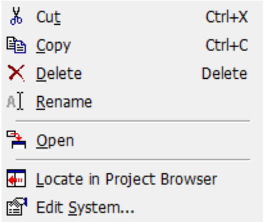
The **Project View** window is displayed when a project is double-clicked in the **Project Browser** window.



Use this window to manage your model on the project level, using the commands on the Visual State Designer menus and toolbars. Typically, you can add and delete projects here.

System context menu

This context menu is available by right-clicking on a system in a **Project View** window:



These commands are available:

Cut, Copy, Delete

Standard Windows editing commands.

Rename

Selects the name of the object so you can edit it.

Open

Opens a **System View** window for the system, see *System View window*, page 294.

Locate in Project Browser

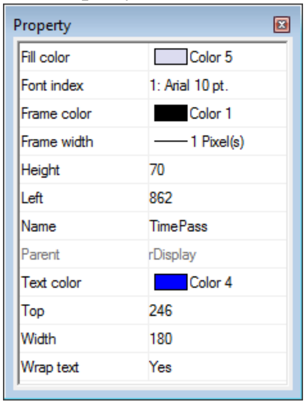
Highlights the object in the **Project Browser** window.

Edit System

Displays the **Edit System** dialog box, see *Edit System dialog box*, page 270.

Property window

The **Property** window is available from the **View** menu.

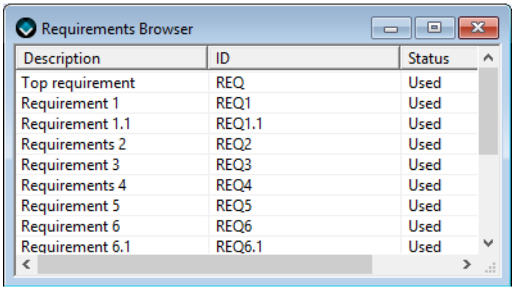


Use this window to specify the properties of objects in the Designer view windows, for example font types for state names, colors of transitions, etc. The contents of the window depends on which objects that are selected in the Designer view windows.

See *Customizing the Designer*, page 235.

Requirements Browser window

The **Requirements Browser** window is available from the **View** menu.

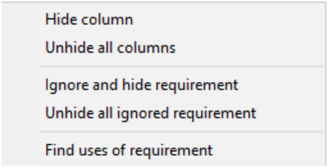


Use this window to inspect the available requirements for your project. The requirements are organized by the attribute definitions used in the imported `.reqif` file. They can only be changed in the file, using an editor or a requirements authoring tool, not in the Designer. Sort the columns by clicking on the column header, or rearrange columns by dragging them.

See *Importing requirements*, page 223.

Requirements context menu

This context menu is available by right-clicking in a column in the **Requirements Browser** window:



These commands are available:

Hide column

Hides the column that you clicked on.

Unhide all columns

Shows all hidden columns again.

Ignore and hide requirement

Makes the requirement that you clicked on unavailable for use and hides it in the window.

Unhide all ignored requirements

Shows all hidden and ignored requirements again, making them available for use.

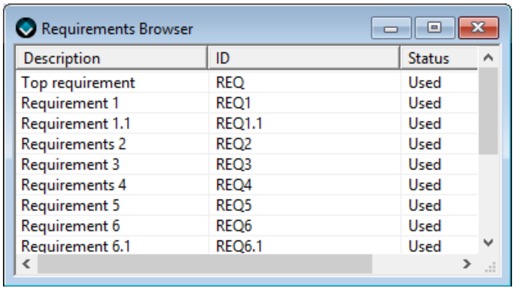
Find uses of requirement

Searches for where the requirement you clicked on is being used, and displays the results on the **Find** tab in the **Output** window.

Note: The information about hidden columns and ignored requirements is stored in the `.vdi` file for the user working with the Designer. Other users can still see all requirements and attributes, and might have their own customized view.

Select Requirements window

The **Select Requirements** window is available from several editing dialog boxes in the Designer.

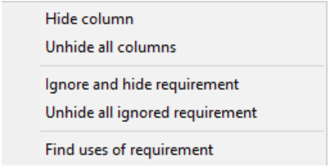


Use this window to tie one or more requirements to an item in your model. The requirements are organized by the attribute definitions used in the imported `.reqif` file. They can only be changed in the file, using an editor or a requirements authoring tool, not in the Designer. Sort the columns by clicking on the column header, or rearrange columns by dragging them.

See *Using requirements files*, page 223.

Requirements context menu

This context menu is available by right-clicking in a column in the **Select Requirements** window:



These commands are available:

Hide column

Hides the column that you clicked on.

Unhide all columns

Shows all hidden columns again.

Ignore and hide requirement

Makes the requirement that you clicked on unavailable for use and hides it in the window.

Unhide all ignored requirements

Shows all hidden and ignored requirements again, making them available for use.

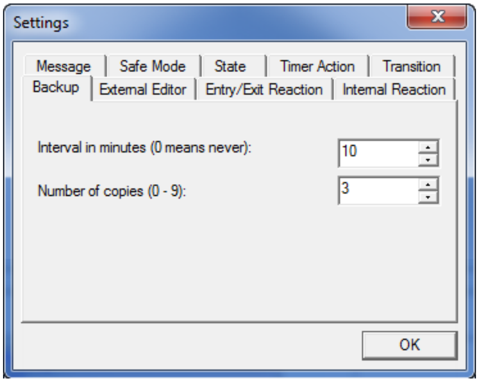
Find uses of requirement

Searches for where the requirement you clicked on is being used, and displays the results on the **Find** tab in the **Output** window.

Note: The information about hidden columns and ignored requirements is stored in the `.vdi` file for the user working with the Designer. Other users can still see all requirements and attributes, and might have their own customized view.

Settings dialog box

The **Settings** dialog box is available from the **Tools** menu.



Use this dialog box to make settings for the Designer. The settings are stored in the registry.

Backup

Use these options to control how the Designer creates backup copies:

Interval in minutes

Use this option to control how often the Designer should back up your model. The backup files created at the given intervals will have the extension `bkt`. There will only be one copy for each file of the interval backup files.

Number of copies

Use this option to set the number of backup copies the Designer should create when a project is saved.

By default, the Designer creates backup files of the `vssm`, `vsp`, and `vsr` files on every save of the project.

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`.

These backup files are created in the same directory where the project is located and you can have up to nine backup files. See *Using Designer backup files*, page 235.

External Editor

Use these options to specify which editor to use:

External source code editor

Specify the editor to use and its path. A browse button is available for your convenience.

Additional command line parameters

Optional: Specify additional command line parameters to send to the editor.

Entry/Exit Reaction

Use these options to make settings for entry/exit reactions:

Use alias names (if defined)

Displays the alias names (if there are any) for entry and exit reactions.

Internal Reaction

Use these options to make settings for internal reactions:

Use alias names (if defined)

Displays the alias names (if there are any) for internal reactions.

Show short state names

Displays abbreviated versions of state names in internal reactions.

Message

Use these options to make settings for messages:

Show messages when deleting/moving elements

Displays a warning message when you delete or move a transition element.

Show delete messages when deleting objects

Displays a confirmation message when you delete a graphical object in a diagram.

Safe Mode

Use these options to enable Visual State safe mode:

Enable safe mode

Enables safe mode, which means you will receive a warning when you create or use a non-verifiable design element.

Show message when

Displays a warning message if you create or use a non-verifiable transition element. Choose between: **Creating unsafe elements**, **Using unsafe elements**, **Creating and using unsafe elements**.

State

Use these options to make settings for states:

Use alias names (if defined)

Displays the alias names (if there are any) for states.

Timer Action

Use these options to make settings for timer actions:

Create timer stop function

Creates a timer stop function automatically (an action function with the name *TimerName*_stop) every time you create a timer action.

Transition

Use these options to make settings for transitions:

Show short state names

Displays abbreviated versions of state names in transitions.

Use alias names (if defined)

Displays the alias names (if there are any) for internal reactions.

Show route points

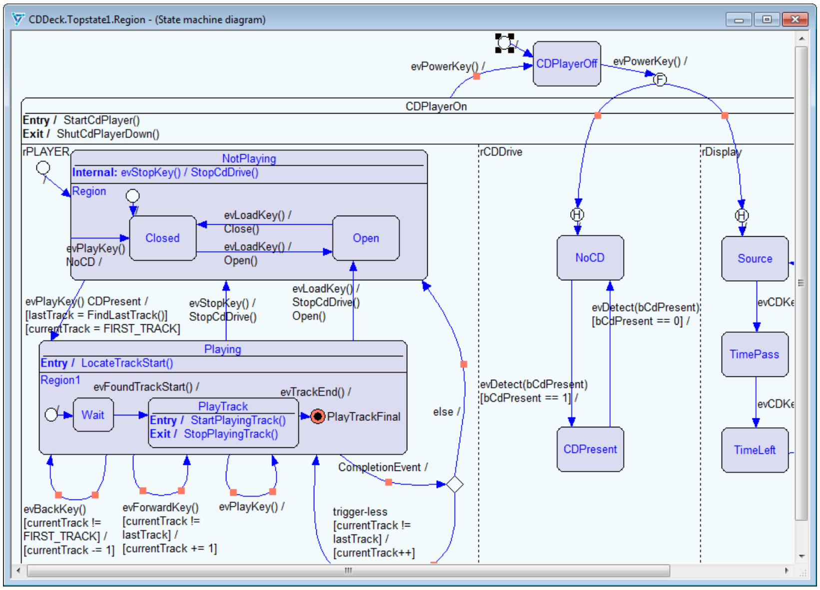
Displays the route points used for manipulating transitions in a diagram also when a transition is not selected.

Auto format orthogonal transitions

Orthogonal transitions () will be automatically drawn after you have clicked the source and destination state for the transition. See also *Creating transitions between your states*, page 130.

State machine diagram window

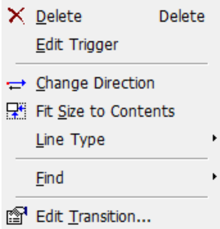
The **State machine diagram** window is displayed when a region is double-clicked in the **Project Browser** window or in the **System View** window.



Use this window to design your model, using the commands on the Visual State Designer menus and toolbars. See also *Designing state machines*, page 126.

Transition context menu

This context menu is available by right-clicking on a transition in the **State machine diagram** window:



These commands are available:

Delete

Deletes the selected object.

Edit Trigger

Makes the name of the transition's trigger, if it has one, editable. If you change the name to the name of an existing event, event group, or signal, you change the trigger to that element.

Change Direction

Inverts the direction of the transition.

Fit Size to Contents

Changes the size of the description box to fit the text.

Line type

Opens a submenu where you can choose which type of line to represent the transition by.

Find

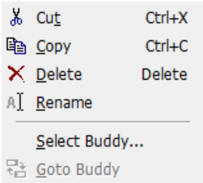
Opens a submenu that contains the items used in this transition, so you can search for other uses of these items.

Edit Transition

Displays the **Edit Transition** dialog box, see *Edit Transition dialog box*, page 272.

Connector state context menu

This context menu is available by right-clicking on a connector state in the **State machine diagram** window:



These commands are available:

Cut, Copy, Delete

Standard Windows editing commands.

Rename

Selects the name of the object so you can edit it.

Select Buddy

Displays a dialog box where you can connect the selected connector state with another connector state.

Go to Buddy

Selects the connected connector state.

Note context menu

See *General Designer windows context menus*, page 298

Standard Designer context menu

See *General Designer windows context menus*, page 298

State context menu

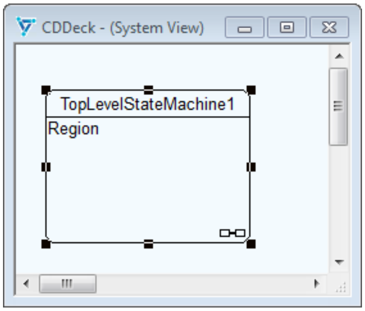
See *General Designer windows context menus*, page 298

System context menu

See *System View window*, page 294

System View window

The **System View** window is displayed when a system is double-clicked in the **Project Browser** window.



Use this window to manage your model on the system level, using the commands on the Visual State Designer menus and toolbars. Typically, you can add and delete top-level state machines here.

Region context menu

See *General Designer windows context menus*, page 298

Note context menu

See *General Designer windows context menus*, page 298

Standard Designer context menu

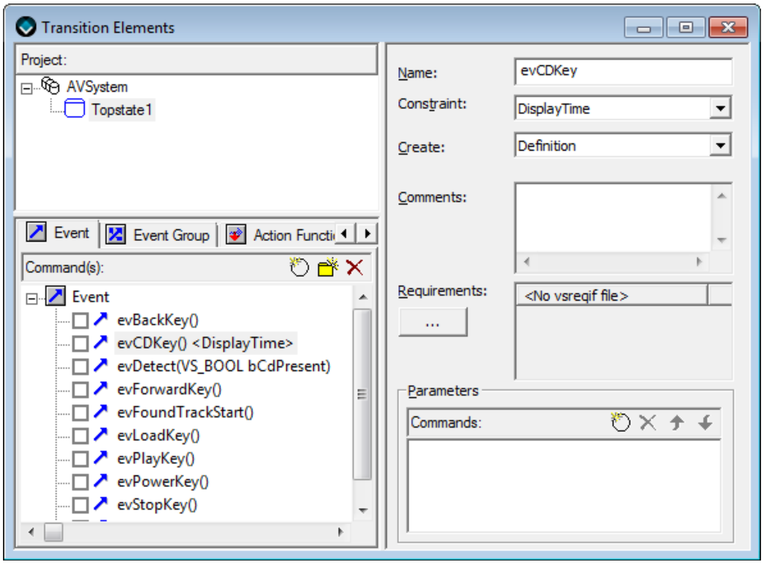
See *General Designer windows context menus*, page 298

State context menu

See *General Designer windows context menus*, page 298

Transition Elements window

The **Transition Elements** window is available from the **View** menu.



Use this window to create, define, edit, rename, and delete transition elements that can be used for creating conditions and actions for states and transitions, and to get a complete overview of all transition elements created for the project.

See *Creating a transition element*, page 184 and *Introduction to transition elements*, page 177.

Project

Displays the project including top-level state machines and state machine templates. To see all global elements, select the project in the tree. To see all local elements for a top-level state machine, select the state that denotes the top-level state machine.

Commands

Displays the individual transition elements. There are nine types of transition elements. Signals and internal variables can only be local, all other element types can be either local or global (except enumerations, which can only be added to transition element files). The available types of transition elements are:

- Event
- Event group
- Action function
- Timer action function
- Signal
- Internal variable
- External variable
- Constant
- Enumeration

Click a category tab to see the available transition elements of that type, for the project or the selected top-level state machine. When you select an element in the list, you can edit it in the editing pane to the right.

Previously created elements are displayed in the **Commands** area and can be dragged from there to the project or top-level state machines in the tree in the **Project** pane. Thus, local elements can become global elements by dragging them to the project in the tree.

There are three buttons available:

New

Creates a new transition element.

New folder

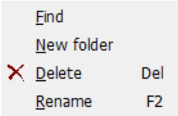
Creates a folder, which means that you can group the transition elements. Click the name to specify a name of your choice.

Delete

Deletes the selected transition element in the list.

Commands context menu

This context menu is available in the **Commands** area:



These commands are available:

Find

Searches for the selected transition element. The search result is displayed in the **Output** window.

New folder

Creates a folder, which means that you can group the transition elements. Click the name to specify a name of your choice.

Delete

Deletes the selected item(s).

Rename

Renames the selected item(s).

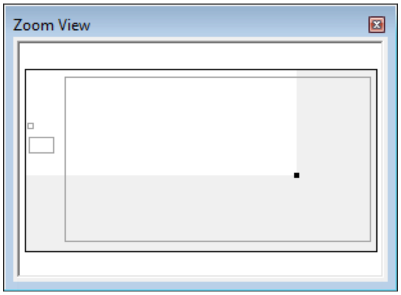
Editing pane

This part of the window differs depending on the selected transition element, see:

- *Edit Event dialog box*, page 248
- *Edit Event Group dialog box*, page 250
- *Edit Action dialog box*, page 244
- *Edit Signal dialog box*, page 260
- *Edit Internal Variable dialog box*, page 255
- *Edit External Variable dialog box*, page 252
- *Edit Constant dialog box*, page 246
- *Edit Enumeration dialog box*, page 247

Zoom View window

The **Zoom View** window is available from the **View** menu.



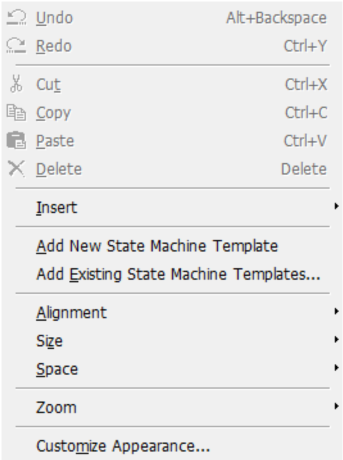
Use this window to see your entire project and the position of your current view in the Designer view windows. See also *Navigating in the state machine diagram*, page 233.

General Designer windows context menus

These context menus are available in several Designer view windows.

Standard Designer context menu

This context menu is available by right-clicking on the background in a Designer view window:



These commands are available:

Undo

Undoes the last edit made in the Designer.

Redo

Redoes the last edit that was undone in the Designer.

Cut, Copy, Paste, Delete

Standard Windows editing commands.

Insert

Shortcuts to all commands on the **Insert** menu, see *Insert menu*, page 308.

Add New State Machine Template

Adds a new state machine template, see *Reusing designs using state machine templates*, page 201.

Add Existing State Machine Templates

Displays a dialog box where you can locate existing state machine templates, see *Reusing designs using state machine templates*, page 201.

Alignment

Shortcuts to all commands on the **Format>Alignment** submenu, see *Format menu*, page 310.

Size

Shortcuts to all commands on the **Format>Size** submenu, see *Format menu*, page 310.

Space

Shortcuts to all commands on the **Format>Space** submenu, see *Format menu*, page 310.

Zoom

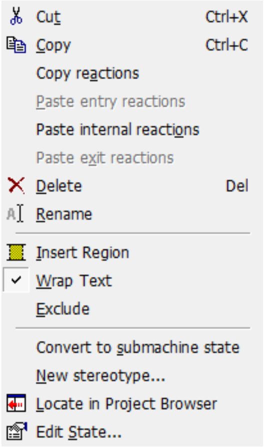
Shortcuts to all zoom commands on the **View** menu, see *View menu*, page 306.

Customize Appearance

Displays the **Customize Appearance** dialog box, see *Customize Appearance dialog box*, page 242.

State context menu

This context menu is available by right-clicking on a state in a Designer view window:



Depending on which type of state you are double-clicking on, some or all of these commands are available:

Cut, Copy

Standard Windows editing commands.

Copy reactions

Copies all reactions from the selected state and stores them on the clipboard.

Paste entry reactions

Pastes all entry reactions from the clipboard to the selected state.

Paste internal reactions

Pastes all internal reactions from the clipboard to the selected state.

Paste exit reactions

Pastes all exit reactions from the clipboard to the selected state.

Delete

Deletes the object you selected.

Rename

Selects the name of the object so you can edit it.

Insert Region

Creates a new region.

Wrap Text

Wraps the text lines to fit within the frame.

Exclude

Excludes the state from further processing. All states or regions that are contained inside the state are also excluded.

Convert to Submachine State

Converts the state to a submachine state, see *Reusing designs using state machine templates*, page 201.

New Stereotype

Creates a new stereotype based on the state you opened the context menu from.

Locate in Project Browser

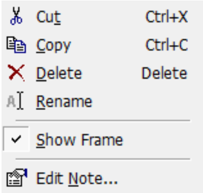
Highlights the object in the **Project Browser** window.

Edit State

Displays the **Edit State** dialog box, see *Edit State dialog box*, page 261.

Note context menu

This context menu is available by right-clicking on a note in a Designer view window:



These commands are available:

Cut, Copy, Delete

Standard Windows editing commands.

Rename

Selects the note text so you can edit it.

Show Frame

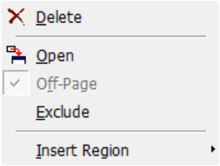
Shows/hides a visible frame around the note.

Edit Note

Displays the **Edit Note** dialog box, see *Edit Note dialog box*, page 256.

Region context menu

This context menu is available by right-clicking on a region in a Designer view window:



These commands are available:

Delete

Deletes the selected object.

Open

Opens a **View** window for the region.

Off-Page

Hides the contents of the region in a separate diagram to make it easier to get an overview of the overall structure of your model.

Exclude

Excludes the region from further processing. All states or regions that are contained inside the region are also excluded.

Insert Region

Creates a new region.

Reference information on Designer menus

This section gives reference information on the menus specific to the Designer. More specifically, this means:

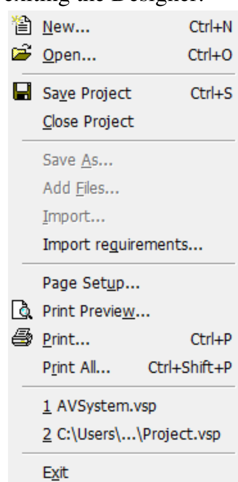
- *File menu*, page 303
- *Edit menu*, page 305
- *View menu*, page 306
- *Insert menu*, page 308
- *Format menu*, page 310
- *Tools menu*, page 311
- *Window menu*, page 312
- *Help menu*, page 313
- *Designer shortcut key summary*, page 313

See also:

- *General Designer windows context menus*, page 298

File menu

The **File** menu provides commands for creating or opening projects and state machine diagrams, importing functions, declarations, and requirements, saving and printing, and exiting the Designer:



Menu commands

These commands are available on the menu:

New (Ctrl+N)

Displays a standard dialog box where you can create a new project.

Open (Ctrl+O)

Displays a standard dialog box where you can open a project or top-level state machine file.

Save Project (Ctrl+S)

Saves the current project.

Close Project

Closes the current project.

Save As

Displays a standard dialog box where you can save the current project or state machine diagram file with a new name.

Add Files

Displays a standard dialog box where you can locate existing files to add to the system.

Import

Imports function declarations and constants contained in a C header file.

Import Requirements

Imports requirements from an existing `.reqif` file into the Designer.

Page Setup

Displays a dialog box where you can set printing options.

Print Preview

Displays a preview of how the printed state machine diagram will look before you print it.

Print (Ctrl+P)

Prints the active state machine diagram.

Print All (Ctrl+Shift+P)

Displays a dialog box where you can choose which parts of the project you want to print.

***filename*.vsp**

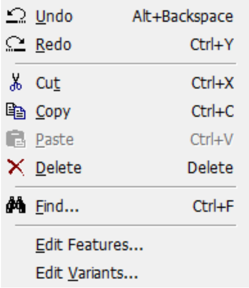
A numbered list of the most recently opened project files. Choose the one you want to open.

Exit

Exits the Designer. You will be asked whether to save any changes to files before they are closed.

Edit menu

The **Edit** menu provides commands for editing.



Menu commands

These commands are available on the menu:

Undo (Alt+Backspace)

Undoes your most recent action. See the status bar for information about what to be undone.

Redo (Ctrl+Y)

Redoes the last edit that was undone in the Designer. See the status bar for information about what to be redone.

Cut (Shift+Del), Copy (Ctrl+C), Paste (Ctrl+V)

The standard Windows commands.

Delete (Delete)

Deletes the selected objects.

Find (Ctrl+F)

Displays a dialog box where you can search for text in projects, systems, and top-level state machines, see *Find dialog box*, page 276.

Edit Features

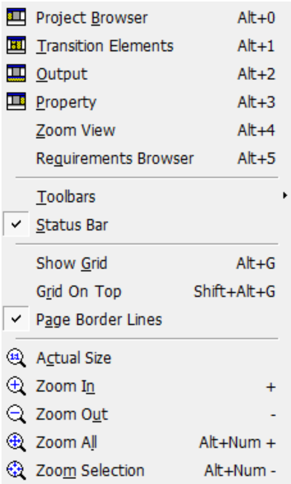
Displays a dialog box where you can define and edit features; see *Edit Features dialog box*, page 253.

Edit Variants

Displays a dialog box where you can define and edit variants; see *Edit Variants dialog box*, page 274.

View menu

The **View** menu provides commands for opening windows, displaying toolbars, and zooming in windows.



Menu commands

These commands are available on the menu:

Project Browser (Alt+0)

Opens the **Project Browser** window, see *Project Browser window*, page 278.

Transition Elements (Alt+1)

Opens the **Transition Elements** window, see *Transition Elements window*, page 295.

Output (Alt+2)

Opens the **Output** window, see *Output window*, page 278.

Property (Alt+3)

Opens the **Property** window, see *Property window*, page 286.

Zoom View (Alt+4)

Opens the **Zoom View** window, see *Zoom View window*, page 298.

Requirements Browser (Alt+5)

Opens the **Requirements Browser** window, see *Requirements Browser window*, page 286.

Toolbars

The commands on this submenu show/hide the Designer toolbars.

Status Bar

Shows/hides the status bar at the bottom of the Designer.

Show Grid (Alt+G)

Shows/hides a grid, that supports you when you draw in the diagrams.

Grid on Top (Shift+Alt+G)

Displays the support grid on top of all objects in the diagrams.

Page Border Lines

Shows/hides the border lines, that define the editable area of the **State machine diagram** window.

Actual Size

Sets the zoom level to 100%

Zoom In (+)

Zooms in on the active diagram to show details better.

Zoom Out (–)

Zooms out in the active diagram to show more objects.

Zoom All (Alt+Numerical +)

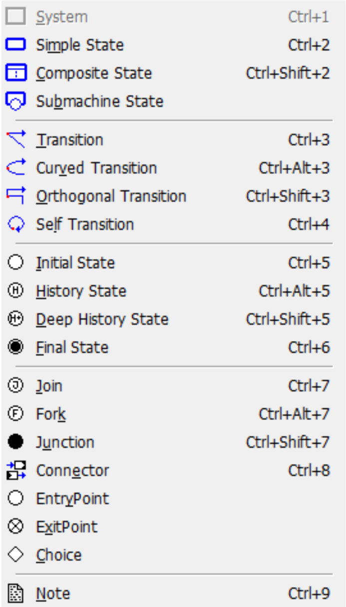
Sets the zoom level so that all objects in the current diagram fit exactly in the view.

Zoom Selection (Alt+Numerical –)

Sets the zoom level so that the selected objects in the current diagram fit exactly in the view.

Insert menu

The **Insert** menu provides commands for inserting graphical objects in the state machine diagrams.



Menu commands

These commands are available on the menu:

System

Inserts a system in the **System View** window. See also *The Visual State system*, page 123.

Simple State

Inserts a simple state in the diagram. See also *Simple state*, page 140.

Composite State

Inserts a submachine state in the diagram. See also *Composite state*, page 140.

Submachine State

Inserts a submachine state in the diagram. See also *State machine templates and submachine states*, page 201.

Transition

Inserts a transition in the diagram. See also *Introduction to transitions*, page 167.

Curved Transition

Inserts a curved transition in the diagram. See also *Introduction to transitions*, page 167.

Orthogonal Transition

Inserts an orthogonal transition in the diagram. Note that if you have selected **Tools>Settings>Transitions>Auto format orthogonal transitions**, the transition will be drawn automatically if you just click the source and then the destination state. See also *Introduction to transitions*, page 167.

Self Transition

Inserts a self transition in the diagram. See also *Introduction to transitions*, page 167.

Initial State

Inserts an initial state in the diagram. See also *Initial state*, page 141.

History State

Inserts a history pseudostate in the diagram. See also *Shallow history pseudostate*, page 143.

Deep History State

Inserts a deep history pseudostate in the diagram. See also *Deep history pseudostate*, page 147.

Final State

Inserts a final state in the diagram.

Join State

Inserts a join pseudostate in the diagram. See also *Join and fork pseudostates*, page 148.

Fork State

Inserts a fork pseudostate in the diagram. See also *Join and fork pseudostates*, page 148.

Junction State

Inserts a junction pseudostate in the diagram. See also *Junction pseudostate*, page 149.

Connector State

Inserts a connector pseudostate in the diagram. See also *Connector pseudostate*, page 149.

Entry Point

Inserts an entry point in the diagram. See also *State machine templates and submachine states*, page 201.

Exit Point

Inserts an exit point in the diagram. See also *State machine templates and submachine states*, page 201.

Choice

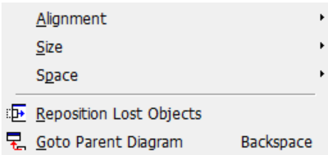
Inserts a choice state in the diagram. See also *Choice state*, page 150.

Note

Inserts a note in the diagram. See also *Inserting notes*, page 232,

Format menu

The **Format** menu provides commands for adjusting the graphical objects in the diagrams.



Menu commands

These commands are available on the menu:

Alignment

Opens a submenu for aligning the selected objects in the active diagram in relation to each other.

Size

Opens a submenu for changing the sizes of the selected objects in the active diagram in relation to each other.

Space

If at least three graphical objects are selected, this command opens a submenu for changing the space between the selected objects in the active diagram.

Reposition Lost Objects

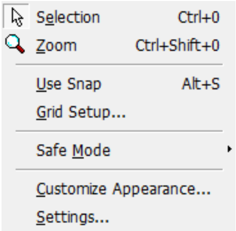
Moves objects located outside the diagram onto the diagram.

Go to Parent Diagram (Backspace)

Changes the active Designer view window to a window one level up in the hierarchy, for example from a **State Machine Diagram View** window to the corresponding **System View** window.

Tools menu

The **Tools** menu provides commands for making settings for working with state machine diagrams.



Menu commands

These commands are available on the menu:

Selection (Ctrl+0)

Toggles the selection tool on/off. If you choose this command when you are using another tool on the **Diagram** toolbar, that tool is deactivated.

Zoom (Ctrl+Shift+0)

Toggles the zoom tool on/off. If you choose this command when you are using another tool on the **Diagram** toolbar, that tool is deactivated.

Use Snap (Alt+S)

Makes graphical objects in diagrams snap to the supporting grid when you move them (including when the grid is invisible).

Grid Setup

Displays the **Grid Setup** dialog box, see *Grid Setup dialog box*, page 277.

Safe Mode>Enable

Enables Safe Mode, which means that you will get a warning when you create and/or use a non-verifiable design element. See *Getting warnings for non-verifiable elements*, page 233.

Safe Mode>Message on Create

Creates a warning when you create a non-verifiable design element.

Safe Mode>Message on Use

Creates a warning when you use a non-verifiable design element.

Safe Mode>Message on Create and Use

Creates a warning both when you create and when you use a non-verifiable design element.

Customize Appearance

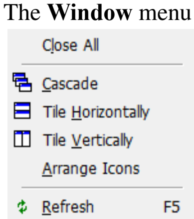
Displays the **Customize Appearance** dialog box, see *Customize Appearance dialog box*, page 242.

Settings

Displays the **Settings** dialog box, see *Settings dialog box*, page 289.

Window menu

The **Window** menu provides commands for arranging the Designer windows.



Menu commands

These commands are available on the menu:

Close All

Closes all Designer view windows.

Cascade

Arranges the open Designer view windows partially on top of each other but fanned out so that the window titles are visible.

Tile Horizontally

Changes the size of the open Designer view windows and arranges them from top to bottom so that they are all visible.

Tile Vertically

Changes the size of the open Designer view windows and arranges them from left to right so that they are all visible.

Arrange Icons

Arranges any icons for minimized windows.

Refresh (F5)

Reloads the contents in the active Designer view window.

Help menu

The **Help** menu provides help for IAR Visual State and displays the version number of the Designer.

Designer shortcut key summary

General

These are the general shortcut keys:

Description	Shortcut key
Create a new project	Ctrl+N
Open a project	Ctrl+O
Save a project	Ctrl+S
Print the active diagram	Ctrl+P
Print all	Ctrl+Shift+P
Make the **Project Browser** window the active window	Alt+0
Make the **Transition Elements** window the active window	Alt+1
Make the **Output** window the active window	Alt+2
Open the **Property** window	Alt+3
Open the **Zoom View** window	Alt+4
Make the **Requirements Browser** window the active window	Alt+5
Refresh the active window	F5
Open the online help system	F1
Close the active window	Alt+F4

Table 16: General Designer shortcut keys

Working with Designer view windows

These are the shortcut keys for working with Designer view windows:

Description	Shortcut key
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
Scroll down one page	Page Down
Scroll left one page	Ctrl+Shift+Left Arrow
Scroll right one page	Ctrl+Shift+Right Arrow
Go to the top of the window	Home
Go to the bottom of the window	Ctrl+Home
Go to the left side of the window	Ctrl+End
Go to the right side of the window	Ctrl+Tab
Zoom in	+
Zoom out	–
Make all objects fit exactly in the view	Zoom + Plus or Alt + Num Plus
Make selected objects fit exactly in the view	Zoom + Minus or Alt + Num Minus
Zoom to 100%	Ctrl+right-click when the Zoom tool is active
Show the grid	Alt+G
Display the grid on top of all objects	Alt+Shift+G
Make objects snap to the grid when you move them	Alt+S

Table 17: Designer view shortcut keys

Editing in diagrams

These are the shortcut keys for editing in diagrams:

Description	Shortcut key
Go to the next graphical object	Tab
Go to the previous graphical object	Shift+Tab
Edit a selected graphical object	Enter
Move a selected graphical object one grid unit	Left/Up/Down/Right Arrow

Table 18: Editing shortcut keys

Description	Shortcut key
Move a selected graphical object one pixel	Shift+Left/Up/Down/Right Arrow
Search for a transition element	Ctrl+F
Activate the selection tool	Ctrl+0
Deactivate an active diagram tool	Right-click
Activate the note tool	Ctrl+9
Activate the zooming tool	Ctrl+Shift+0
Shift focus to a parent diagram	Backspace
Activate the insert standard transition tool	Ctrl+3
Activate the insert curved transition tool	Ctrl+Alt+3
Activate the insert orthogonal transition tool	Ctrl+Shift+3
Clone an existing graphical object	Ctrl+drag the object
Activate the insert simple state tool	Ctrl+2
Activate the insert composite state tool	Ctrl+Shift+2
Define the number of regions in a composite state	Ctrl+draw a composite state
Change places for two regions with a state	Shift+drag a region
Activate the insert initial state tool	Ctrl+5
Activate the insert shallow history state tool	Ctrl+Alt+5
Activate the insert deep history state tool	Ctrl+Shift+5
Activate the insert final state tool	Ctrl+6
Activate the insert join state tool	Ctrl+7
Activate the insert fork state tool	Ctrl+Alt+7
Activate the insert junction state tool	Ctrl+Shift+7
Activate the insert connector state tool	Ctrl+8

Table 18: Editing shortcut keys

Editing transition elements shortcut keys

These are the shortcut keys for editing transition elements:

Description	Shortcut key
Create a new element	Ctrl+N
Select the next element type	Ctrl+Page Down
Select the next element type	Ctrl+Page Up

Table 19: Editing transition elements shortcut keys

Syntax of C header files

The import functionality recognizes most C and C++ constructs from header files, so many header files that are used as part of your ordinary projects can be imported to the Designer.

In addition to the regular syntax for function declarations and constant definitions from source files, some extra special syntax is supported for importing other items.

SYNTAX FOR IMPORT OF FUNCTION DECLARATIONS

Import of function declarations (map to action functions in Visual State) can be done either by a single import statement or multiple import statements.

Single import statement

```
#pragma VS_ACTION function_declaration
```

where *function_declaration* is a standard Standard 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 Standard C function declaration.

SYNTAX FOR IMPORT OF CONSTANTS

Import of constants (map to constants in IAR Visual State) is done by multiple import statements as follows:

```
#pragma VS_CONSTANT_BEGIN
  macro_statement 1
  ...
  macro_statement N
#pragma VS_END
```

where *macro_statement 1 ... N* is a standard Standard C macro statement.

In addition to the standard syntax, you can specify which Visual State type the constant should have by inserting a typecast to the desired type, for example like this:

```
#define BOOLValue1 (VS_BOOL)1
#define BOOLValue2 ((VS_BOOL)1)
#define DoubleValue1 ((double)-7)
#define DoubleValue2 (VS_DOUBLE)1.0
```

This is an example of the 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
```

SYNTAX FOR IMPORTING TRIGGERS

Import triggers (events and signals in IAR Visual State) by a multiple import of statements like this:

```
#pragma signal lowBattery
#pragma signal lowFuel BAEA6E65-7AFC-45EE-86FC-78E86A48CC87
#pragma event horn
#pragma event wiper(int intensity)
        A11A0B3B-F590-48D3-9D83-572C5D3665AF
```

In this example, the individual lines will import:

- A new signal with the name `lowBattery`
- A new signal with the name `lowFluel` and the given GUID
- A new event with the name `horn`
- A new signal with the name `wiper`, taking one argument of type `VS_INT` named `intensity`, and with the given GUID.

Be careful when you specify GUIDs for imported items. A GUID must be unique within the model.

Part 4. Simulating using the Validator

This part of the *IAR Visual State User Guide* includes these chapters:

- Simulation
- Graphical animation
- Tracing
- Analyzing
- Recording and playing test/event sequences
- The Visual State Validator

Simulation

Introduction to simulating your model using the Validator

Learn more about:

BRIEFLY ABOUT SIMULATING USING THE VALIDATOR

Using the Visual State Validator you can simulate your state machine model, which means that you can perform functional testing—check that your application is in accordance with your requirement specification—and validate your state machine model to get insight in the behavior at specific points of execution.

The Validator supports:

- Interactive simulation, including use of conditional breakpoints—you manually send events to one or more systems and view the system's reaction to this, including variables assigned a new value, generated signals, actions, and state changes in the simulated system.

  You can view your interactive animation graphically in the Designer, see *Graphical animation*, page 335.
- Automatic simulation—you test your model automatically by applying a test sequence of events and assignments that you have recorded to a *sequence file*. See *Recording and playing your test sequences*, page 350.
- Tracing—that is, to obtain a sequence of events that will get the system into a desired configuration. See *Tracing state machine models*, page 341.
- Listing the Visual State elements used, and test coverage. See *Analyzing using the Validator*, page 346.

In addition to using the Validator for simulation, you can also use it for testing your state machine model in a target application by means of RealLink, see *Debugging design models using RealLink*, page 785.

Note: If you are using IAR Visual State together with IAR Embedded Workbench, use C-SPYLink instead of RealLink, see *Debugging design models using C-SPYLink*, page 759.

DEBUGGING MODES

The Validator has two debugging modes:

Validator mode	In this mode, you simulate your design model and monitor how it behaves.
Target mode	When the Validator is connected to a target by means of RealLink, you can see the values as they are on target. Use the command **Show target values** (available from context menus in the Validator windows) to switch between showing values as they would be on target and as they would appear during simulation.

See *Toggling between Validator mode and Target mode for a window*, page 334.

VIEWING ELEMENTS DURING SIMULATION

When an event has been sent, a number of elements will be affected. Via the Validator windows, you can track changes in the following elements:

- States

  The states that became active upon sending an event, and the states that were current before the event was sent, can be viewed in the **Systems** window. Current states are shown with a red arrow. See also *Systems window*, page 389.

- Actions

  Actions, or output, produced by the sent event are listed in the **Action** window, which also lists the arguments with which the actions were called. See *Actions window*, page 364.

  The order in which output is listed is runtime-specific, which means that the top-most output was the first output given. This applies to systems, too, if your project contains more than one system. Every time a deduction (microstep) is started for a specific system/instance, the **Action** window is cleared for output coming from that system/instance, and every time output is given during deduction (microstep), the output is added to the end of the list. For information about microsteps, see *Runtime behavior—macrosteps and microsteps*, page 122.

- Assignments

  Assignments resulting from the sent event are displayed in the **Action** window. See *Actions window*, page 364.

- Signals

  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 retrieved next and the last signal listed is the last added signal (FIFO, first in, first out). If your system is using signals, the **Signal Queue** window displays the signal queue. Note that if the signal queue for a specific system/instance is not empty, it is not possible to send an event to that system/instance. See also *Signal Queues window*, page 387.

  Handling of the signal queue can be automatic or manual:

  - *Automatic signal queue handling*—the queue of a specific system is emptied just after the deduction of a sent event has been performed, and before the event is sent to any other enabled systems. See *Activating automatic signal queue handling*, page 328.
  - *Manual signal queue handling*—the queue is not emptied until event deduction has been completed for all enabled systems. See *Using manual emptying of signal queues*, page 328.

  Note that if the project contains more than one system and when simulating in Validator mode, there is a significant difference between the two approaches to emptying the queue. Also, if assignments are used, the different approaches might give different results.

- Guard expressions

  At runtime, a guard expression is evaluated during deduction. Consequently, the expression can only have the value TRUE or FALSE. However, the **Guard Expressions** window provides a view of expression values between deductions. This means that a guard expression can also have the value N/A (not available). The expression will have this value if any unresolved variables, action functions, or event parameters are included in the guard expression. If any unresolved guard expression is met during a deduction, a dialog box will be displayed where you can specify the value of the unresolved guard. See *Guard Expressions window*, page 380.

- Defined elements

  To view all variables, all action functions, and all constants in all systems, use the **Variables** window. Via the context menu you can show or hide a specific group of elements, show or hide all variables, and filter the information. For arrays, you can choose to display the array indexes. See *Variables window*, page 395.

CONDITIONAL BREAKPOINTS

During simulation, you can set breakpoint conditions on systems for one or more of the following:

- The sent event or signal

- An expression—can be evaluated either before or after a deduction
- The current state, before the deduction
- The next state, after the deduction
- The actions executed during a deduction.

Note: Breakpoints are not available in target mode.

If more than one condition is defined for a breakpoint, they must all be fulfilled before the break is triggered.

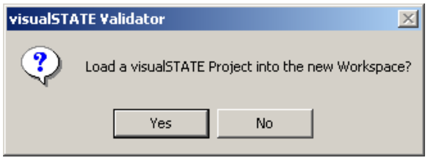
Simulating models using the Validator

What do you want to do?

- *Creating a new Validator workspace*, page 324
- *Preparing for the simulation*, page 325
- *Specifying event parameters*, page 326
- *Sending events manually*, page 327
- *Filtering events*, page 328
- *Activating automatic signal queue handling*, page 328
- *Using manual emptying of signal queues*, page 328
- *Handling signal queues for a single system*, page 329
- *Defining breakpoints*, page 329
- *Using breakpoints*, page 330
- *Changing values of variables*, page 332
- *Setting action function return values*, page 332
- *Forcing states*, page 333
- *Specifying the order of the systems/instances*, page 333
- *Toggling between Validator mode and Target mode for a window*, page 334

CREATING A NEW VALIDATOR WORKSPACE

1 In the Validator, choose **File>New Workspace**.



2 Click **Yes**.

3 In the **Open Project** dialog box that is displayed, specify the project to load.

The selected project will be opened in the workspace.

4 Choose **File>Save Workspace** to save your workspace.

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

PREPARING FOR THE SIMULATION

Before you can start the simulation and send events to the system, you must:

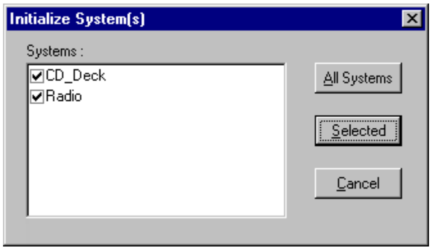
- Initialize the loaded systems
- Send the reset event
- Specify event parameters.

1 In the Navigator, open your workspace and then start the Validator. For example the `CD Player` example project, which you can find in the Information Center. The Validator starts with the workspace that contains the project that you want to simulate.

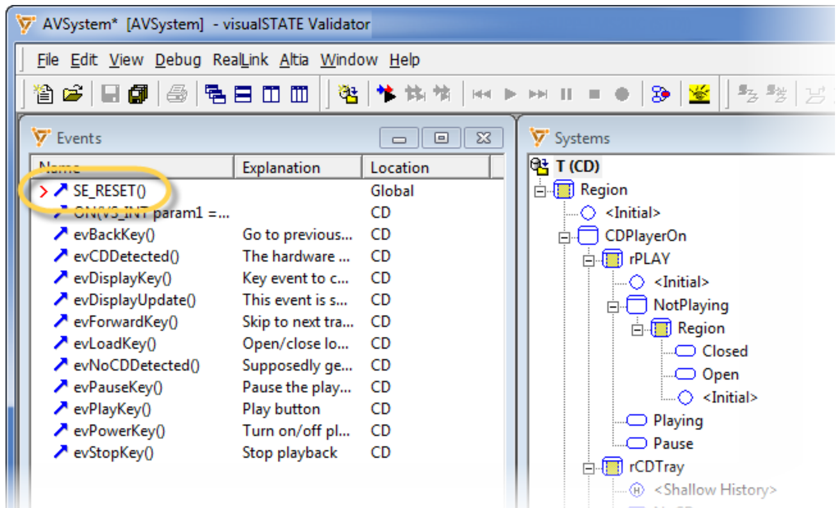
2 Choose **Window>Classic Simulation**.

3 On the **Debug** toolbar, click the **Initialize System(s)** button ( ) to initialize the system.

If your project contains more than one system or system instance, a dialog box is displayed where you can select the systems to initialize:



4 In the **Events** window, double-click `SE_RESET` to send the Visual State reset event to the systems. Note that the reset event is always `SE_RESET` and cannot be changed.



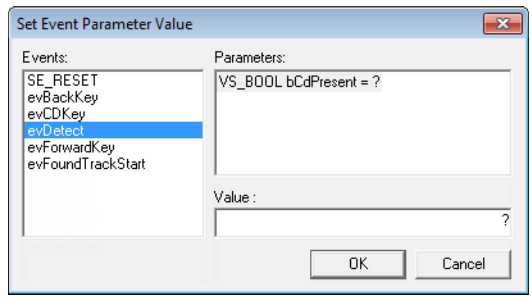
Active events will be marked by a red > mark.

Before you can send your events, specify event parameters.

SPECIFYING EVENT PARAMETERS

If the event has parameters, they must all be assigned a value before it is possible to send the event. The reason is that the event parameters might 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.

1 In the **Event** window, right-click and choose **Set Parameter Values** from the context menu. The **Set Event Parameter Value** dialog box is displayed.



2 Specify the event parameter values, either in the **Value** text field or by editing the parameter directly.

Note: When you use the Validator in target mode, the event parameters for a target event can only be modified using the **Watch** window.

You are now ready to start the simulation by sending events to the loaded systems.

SENDING EVENTS MANUALLY

When you have completed the steps described in *Preparing for the simulation*, page 325, you can start the simulation by sending events to the loaded systems.

1 In the **Events** window, double-click the event to send. You can send *active events*, marked with a red > mark (unless the project contains more than one system and a global event is active in more than one system, in which case it is marked with a red >> mark).

Global events will be sent to all enabled systems. Local events will be sent to the system in which they are defined.

2 When you have sent the event, new events might become active. As a consequence of sending the event, the various Validator windows reflect what happens to actions, states, events, variables, etc.

3 To have guard expressions resolved during the inquiry on active events, right-click in the **Events** window and choose **Include Guard Expressions** from the context 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 make an event inactive.

Note: The **Include Guard Expressions** option is only available in Validator mode (not target mode) because the inquiry on active events by the Visual State API can only check state conditions. See also *Guard expressions*, page 169.

4 You can also use the **Watch** window for sending and viewing events. In the **Events** window, right-click the event and choose **Add to Watch** from the context menu. In the **Watch** window, select the event and press Enter.

FILTERING EVENTS

There are various ways to filter events:

- To hide an event from the **Events** window, right-click the event and chose **Hide** from the context menu.
- To display all hidden events, right-click in the **Events** window and choose **Show All** from the context menu.
- To view only the active events, right-click in the **Events** window and choose **Only Active Events** from the context menu.

ACTIVATING AUTOMATIC SIGNAL QUEUE HANDLING

1 In the Validator, choose **Window>New Window>Signal Queues** to open the **Signal Queues** window.

2 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 might be added to the signal queue.

When automatic signal queue handling is used, microsteps are not available in Target mode.

Note: The system might be in a live lock, meaning that the signal queue will never be emptied. If a live lock occurs, press Escape to stop sending signals. In Target mode, a live lock cannot be stopped.

USING MANUAL EMPTYING OF SIGNAL QUEUES

There are two ways to manually empty the signal queue:

- Continuing to send the top signal in the queues until the queue is empty. To do this, double-click the signal in the **Signal Queues** window.

- Single-stepping the queue. To do this, right-click in the **Signal Queues** window and choose **Send Signal** from the context menu. This will send the top signal in the first queue that contains signals. The order in which the queues are emptied is defined in the **System setup** window, see *System Setup window*, page 391.

HANDLING SIGNAL QUEUES FOR A SINGLE SYSTEM

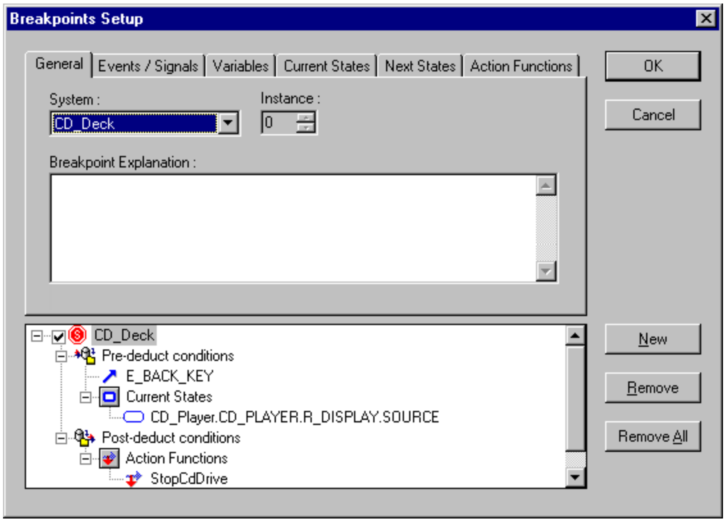
1 In the Validator, choose **Window>New Window>Signal Queues** to open the **Signal Queues** window.

2 To handle the signal queue:

- To empty the signal queue for a specific system, right-click the system and choose **Empty System Signal Queue** from the context menu.
- To step the signal queue for a specific system, right-click the system and choose **Send System Signal** from the context menu.

DEFINING BREAKPOINTS

1 In the Validator, choose **Edit>Breakpoints** to open the **Breakpoints Setup** dialog box:



2 On the **General** page, select the system and instance on which the break should be triggered. Optionally, specify a description for the breakpoint. Click **New** to create the breakpoint.

3 At the bottom of the dialog box you get an overview of all defined breakpoints, including your newly created breakpoint. To enable and disable a breakpoint, select or deselect it.

Note: You can also enable and disable breakpoints in the **Breakpoints** window; choose **View>Breakpoints** to open the window.

4 To set up a breakpoint condition, click the tab for the appropriate condition type (event/signals, variables, etc), and make the appropriate settings on that page.

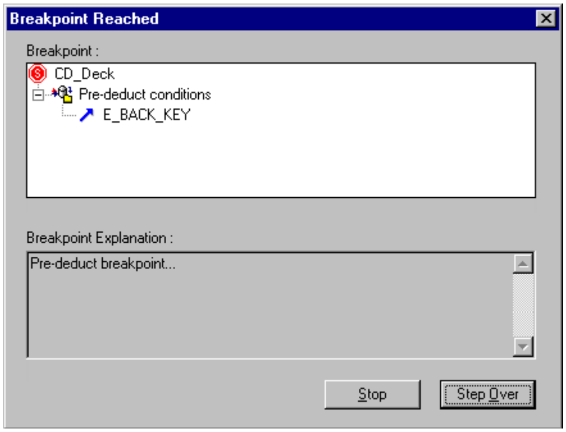
For information about the options, see *Breakpoints Setup dialog box : General*, page 371.

5 To remove a breakpoint, select it and click **Remove**. Alternatively, click **Remove All**.

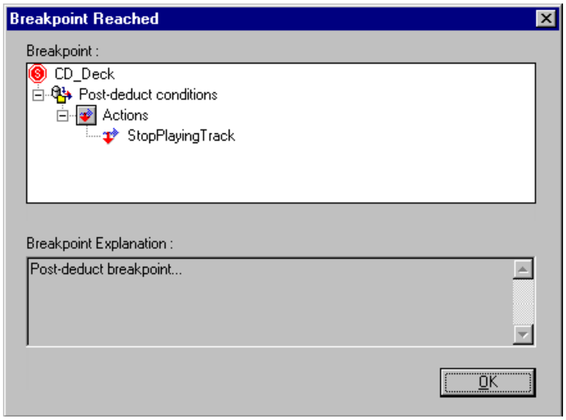
USING BREAKPOINTS

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 is displayed.

1 Click either **Step over** to step over the breakpoint and thereby perform the deduction. Or click **Stop**.



2 After deduction, all post-deduct conditions are evaluated. If all post-deduct conditions in a breakpoint are fulfilled (including all post-deductions), a break is performed. The **Breakpoint Reached** dialog box is displayed:



Click **OK**.

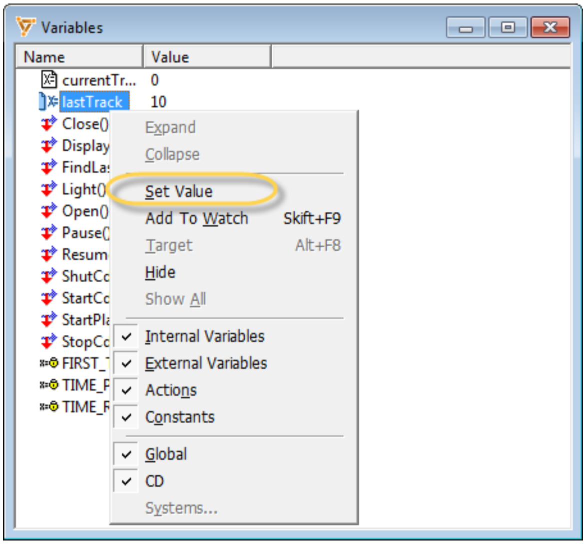
For example, this means that the deduction on the first system will remain if these conditions are fulfilled:

- the project contains two systems
- a deduction has been performed on the first system
- a pre-deduct breakpoint is reached on the second system
- you stop.

Note: If the project contains more than one system/instance, and you stop on a breakpoint, all further processing is disabled.

CHANGING VALUES OF VARIABLES

1 In the Validator, choose **Window>New Window>Variables** to open the **Variables** window.



2 Right-click the variable for which to change the value and choose **Set Value** from the context menu.

Note: At load time, the variables are assigned their initialization values.

Note: Arrays must be expanded before you can set the value of the various indexes.

SETTING ACTION FUNCTION RETURN VALUES

An action function can have a return value. To simulate the system, an action function return value might be necessary if the value is used in a guard expression or an assignment expression.

1 In the Validator, choose **Window>New Window>Variables** to open the **Variables** window.

2 In the **Value** column, select the action function return value, and type the value.

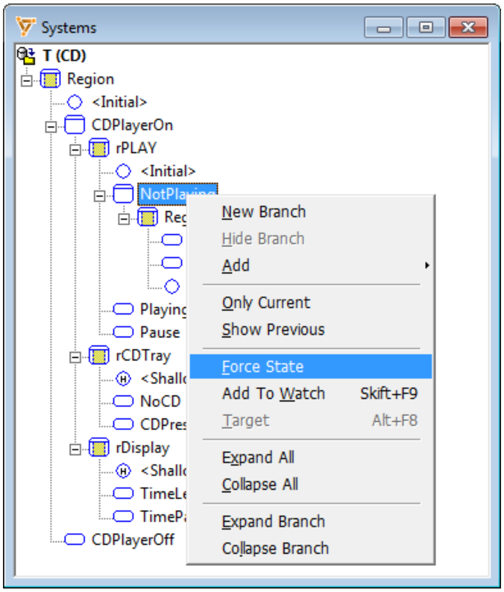
Each time an action function is used you can be prompted to specify a return value. To specify the value, choose **Debug>Action Function Return Value Prompt**. By default, the value is undefined.

Note: In target mode, you cannot view the action function return values.

FORCING STATES

You can force the system to a specific state. All states can be forced. Typically, this can be useful if you want to start from a specific state instead of starting from the beginning; specifically for long execution sequences.

1 In the Validator, choose **Window>New Window>Systems** to open the **Systems** window.



2 Right-click the state to force and choose **Force State** from the context menu.

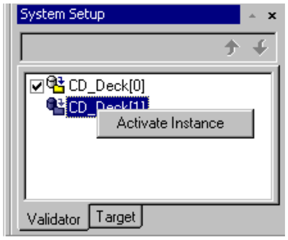
The state is now active in the system.

SPECIFYING THE ORDER OF THE SYSTEMS/INSTANCES

Changing the order of the systems changes the order of how events are sent to the various systems. Thus, it is possible to match how the target application as closely as possible. Furthermore, changing the system order affects how signal queues are handled. If manual signal queue handling is used, the system setup determines which queue that is emptied first. See also *Using manual emptying of signal queues*, page 328.

Note: The system order only applies to interactive simulation (thus, not using test sequence files). When a recorded test sequence is played, all input to the systems is performed on a system/instance basis, and it makes no sense to empty a signal queue manually. See *Recording and playing test/event sequences*, page 349.

1 In the Validator, choose **View>System Setup** to open the **System Setup** window.



2 On the Validator page, change the system order by clicking the Up Arrow or Down Arrow buttons on the toolbar.

3 To enable or disable a system, click the check boxes to the left of the system name. Disabled systems will not receive events.

4 To activate an instance (only possible in Validator mode), right-click the system and choose **Activate Instance** from the context menu.

Note: It is not possible to change instances in target from the Validator.

TOGGLING BETWEEN VALIDATOR MODE AND TARGET MODE FOR A WINDOW

When the Validator is connected to a target by means of RealLink (Target mode), you can change the mode of the windows to view your model in Validator mode or in Target mode. This is possible for all windows, except for the **Guard Expressions** window. See also *Debugging design models using RealLink*, page 785.

1 In the Validator, open your workspace.

2 Open a window, for example the **Events** window.

3 Right-click in the window and choose **Show target values** from the context menu (or press Alt+F8).

The window is now in Target mode.

Graphical animation

- Introduction to graphical animation of debug sessions
- Animating debug sessions graphically
- Graphical environment for graphical animation

Introduction to graphical animation of debug sessions

Learn more about:

- *Graphical animation of debug sessions*, page 335

GRAPHICAL ANIMATION OF DEBUG SESSIONS

Using IAR Visual State you can animate your debug session graphically—while simulating using the Validator, while debugging in Target mode using RealLink, or while debugging on hardware using C-SPYLink.

Regardless of which debugging solution you are using, you can view your animation graphically in the Designer.

Animating debug sessions graphically

What do you want to do?

- *Animating your debug session graphically*, page 335
- *Setting breakpoints for graphical animation*, page 336
- *Customizing shapes and colors for graphical animation*, page 336

ANIMATING YOUR DEBUG SESSION GRAPHICALLY

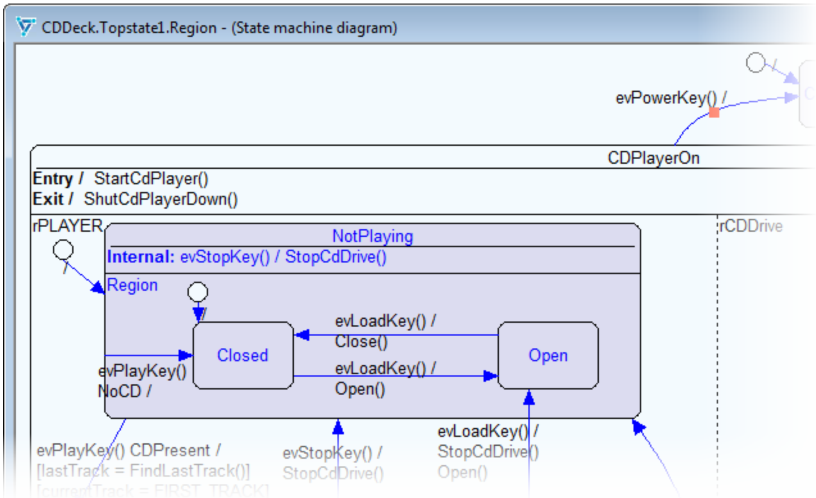
You can view the simulation graphically in the Designer. When you use the Designer for graphical animation, the title bar indicates this (*Simulation*), and you cannot change the design in this Designer instance.

1 Starting graphical animation:

- In the Validator, choose **Debug>Graphical Animation**, or click the **Graphical Animation** button on the toolbar. The Designer starts a graphical animation session.

- In C-SPYLink, choose **Visual State>View>Graphical animation**. The available systems are listed on the submenu; choose the system for which you want to start graphical animation. You can start animation for several systems in parallel.

2 In the Designer, click the system in the **System View** to open the **State machine diagram** window.



When you send an event in the Validator that fires a transition, the affected states and transitions can be viewed in the Designer. All open diagrams are updated each time a microstep is completed. For information about macrosteps, see *Runtime behavior—macrosteps and microsteps*, page 122.

SETTING BREAKPOINTS FOR GRAPHICAL ANIMATION

1 In the Validator, start the Designer for graphical animation. See *Animating your debug session graphically*, page 335.

2 In the Designer, right-click the state for which you want to set a breakpoint and choose **Insert/remove current state breakpoint** or **Insert/remove next state breakpoint** from the context menu.

To delete all breakpoints, choose **Remove all breakpoints**.

CUSTOMIZING SHAPES AND COLORS FOR GRAPHICAL ANIMATION

1 In the Validator, open your workspace.

2 Choose **Debug>Graphical Animation** to start the Designer in simulation mode.

3 In the Designer, choose **Tools>Configure** to open the **Customize Appearance** dialog box.

4 In the dialog box that is displayed, set the shape and color of frame borders and specify whether previous states should be displayed in the simulation diagram.

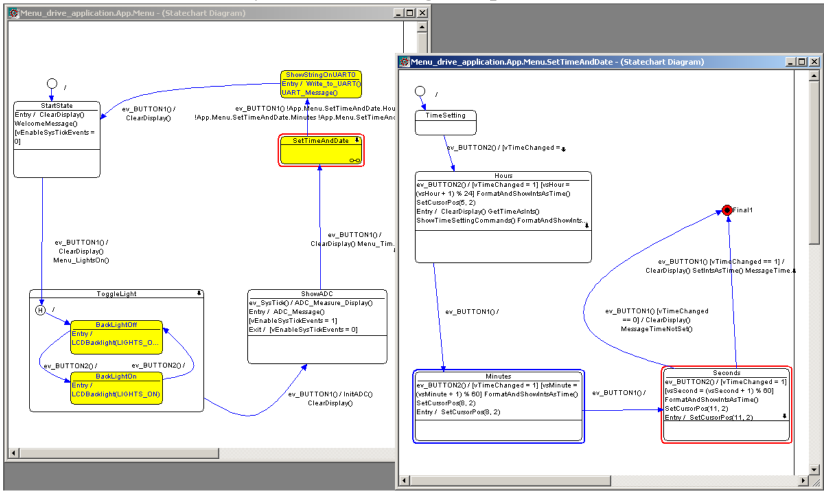
Graphical environment for graphical animation

Reference information about:

- *Designer windows in Graphical Animation mode*, page 337
- *Customize Graphical Animation dialog box*, page 338

Designer windows in Graphical Animation mode

The Designer windows show graphical animation when you choose **Visual State>View>Graphical animation** and choose a system in the IAR Embedded Workbench IDE, or when you choose **Debug>Graphical Animation** in the Validator.



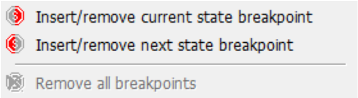
These example windows display an animation of the execution of the state machine directly in the original diagram, as it looks in the Visual State Designer. This feature can be active for the specific system or systems you choose.

Red states indicate newly entered states. Blue states indicate states that were left as the result of the last event processing.

See also *Animating your debug session graphically*, page 335.

Context menu

This context menu is available by right-clicking on a state in the Designer when the Designer is running graphical animation:



These commands are available:

Insert/remove current state breakpoint

Inserts or removes a current state breakpoint from the selected state.

Insert/remove next state breakpoint

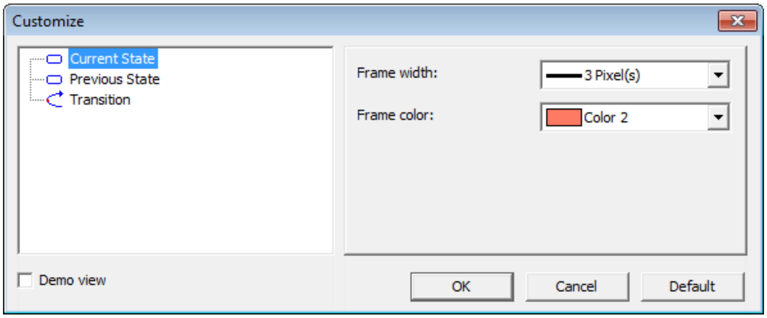
Inserts or removes a next state breakpoint from the selected state.

Remove all breakpoints

Removes all breakpoints in the state machine diagram.

Customize Graphical Animation dialog box

The **Customize Graphical Animation** dialog box is available from the **Tools** menu in the Designer when the Designer is running graphical animation.



Use this dialog box to customize the appearance of states and transitions during the graphical animation. Select the graphical object you want to customize and set these options:

Frame width

Select the width of the state frame.

Frame color

Select the color of the state frame.

Show previous current state

Indicates the previous current state with an extra border around the frame.

Flash fired transitions

Indicates fired transitions by making them flash.

Demo view

Displays a small window with a preview of the current customizations.

Tracing

Introduction to tracing your state machine model

Learn more about:

- *Tracing using the Validator*, page 341

TRACING USING THE VALIDATOR

A trace sequence is a sequence of steps that leads to a desired state configuration. Tracing can be used for answering the question *How do I get from the initial state to a user-defined state 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 Validator's capability for handling sequence files. For information about how to use the resulting test sequence, see *Playing your recorded test sequence*, page 352.

The Verificator is used for finding the actual trace. In a trace, the Verificator will find a suitable sequence of events and external variables values that makes it possible to reach the desired configuration.

Tracing state machine models

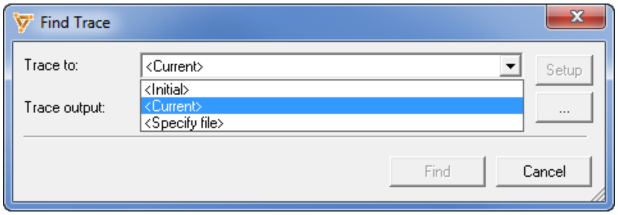
What do you want to do?

- *Setting up a trace*, page 341
- *Setting up the trace point*, page 343

SETTING UP A TRACE

1 In the Validator, open your Validator workspace.

2 Choose **Debug>Find trace** to open the **Find Trace** dialog box:



From the **Trace to** drop-down list, choose between:

Initial — Traces to the initial state in the system. Completes step 2 in this procedure.

Current — Traces to the current state in the system. Completes step 2 in this procedure.

Specify file — Traces to the point specified in a file. Choose this option if you want to specify a customized setup. This means that you must set up the desired configuration of states. Click **Setup**, see *Find Trace dialog box*, page 379.

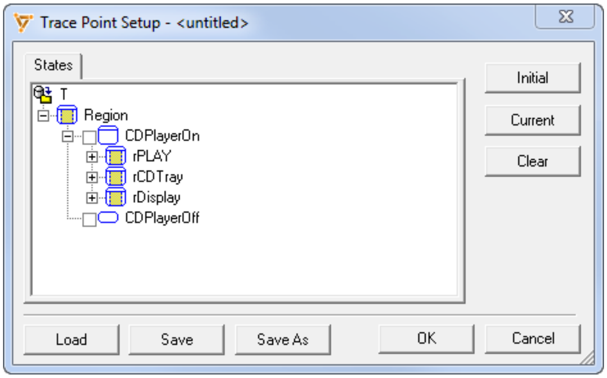
3 Specify a destination file for the output in the **Trace output** text field or use the Browse button.

4 Click **Find** to start tracing to the specified state configuration (trace point). If a trace sequence is found, the resulting sequence is saved to the file you specified.

SETTING UP THE TRACE POINT

A trace point represents the state configuration you want to reach. You can set up your trace point, open an existing configuration, and edit it.

1 When you have selected **Specify file** in the Trace Setup file and clicked **Setup** (as described in *Creating a new Validator workspace*, page 324), the **Trace Point Setup** dialog box is displayed.



2 Select your desired trace point by selecting a state or click one of the buttons:

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.

Use the **Clear** button to clear the state configuration.

3 If your project contains more than one system, choose a system from the **Select System** drop-down list.

4 Use the standard **Load**, **Save**, and **Save As** for loading, saving and renaming your trace point file.

5 When you are finished, click **OK**.

The saved trace point file will be saved with information about 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 because 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.

Analyzing

- Introduction to analyzing using the Validator
- Analyzing using the Validator

Introduction to analyzing using the Validator

Learn more about:

- *Static and dynamic analysis*, page 345

STATIC AND DYNAMIC ANALYSIS

You can use the Validator to analyze your model with regard to element use and test coverage—*static analysis* and *dynamic analysis*, respectively.

Static analysis

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.

See *Performing static analysis*, page 346.

Dynamic analysis

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, dynamic analysis will describe which parts of the model that have the highest activity level, and which parts that are never entered. This information is useful when analyzing how the dynamics of the application will perform at runtime.

See *Performing dynamic analysis*, page 347.

Analyzing using the Validator

What do you want to do?

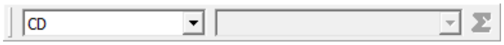
- *Performing static analysis*, page 346
- *Performing dynamic analysis*, page 347

PERFORMING STATIC ANALYSIS

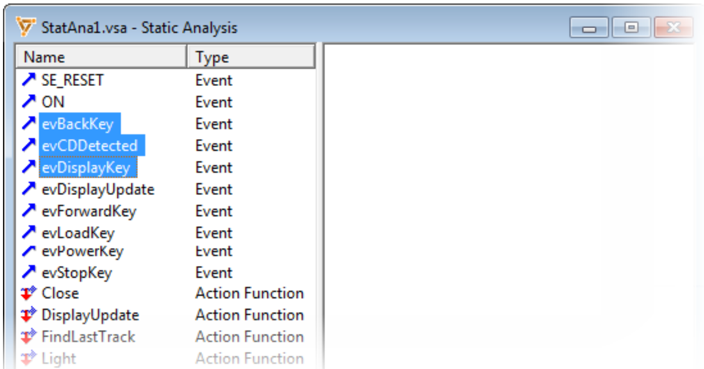
1 In the Validator, open your Validator workspace.

2 Choose **File>Analysis>New Static** to open the **Static Analysis** window.

3 On the **Analysis** toolbar in the Validator main window, select the system on which to perform the analysis, in this example `CD`:

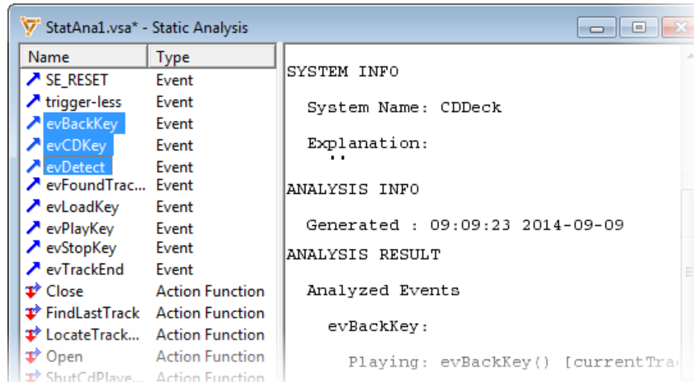


4 In the left pane of the window, select the elements that you want to analyze transitions for:



5 Choose **Debug>Analyze** or click the **Analyze** button on the **Analyze** toolbar.

Analysis will be performed and the results is displayed in the right pane of the window:



6 To save the analysis results, choose **File>Analysis>Save**. Save your file (filename extension `vsa`).

To open an existing static analysis file, choose **File>Analysis>Open**.

PERFORMING DYNAMIC ANALYSIS

1 In the Validator, open your Validator workspace. Initialize the system and send events to the system by double-clicking them. See *Sending events manually*, page 327.

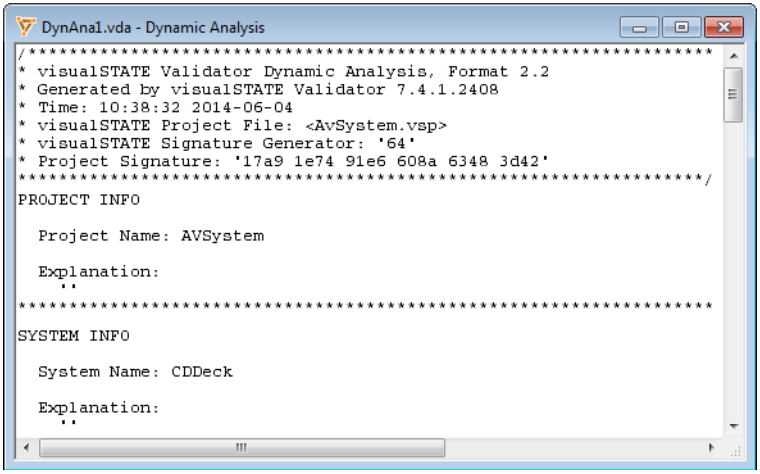
2 Choose **File>Analysis>New Dynamic** to open the **Dynamic Analysis** window.

3 On the **Analysis** toolbar, select the system on which to perform the analysis.

4 On the **Analysis** toolbar, select the sequence for which to perform the analysis. This can be a test sequence in a 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`. 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 the result is displayed in the **Dynamic Analysis** window.



The dynamic analysis consist 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 you can see 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 analysis can be in either text format or comma-separated values format (CSV). You can select which format to use from the context menu.

Note: The dynamic analysis data is reset each time an analysis is performed, and each time **Edit>Undo** is applied to a **Send Event** or a **Send Signal** command.

6 To save the dynamic analysis file, choose **File>Analysis>Save**. Save your file (filename extension `vda`).

7 To open an existing dynamic analysis file, choose **File>Analysis>Open** and specify the file to open.

Recording and playing test/event sequences

Introduction to recording and playing test sequences

Learn more about:

- *Briefly about recording test and event sequences*, page 349
- *Briefly about playing recorded test sequences*, page 350

BRIEFLY ABOUT RECORDING TEST AND EVENT SEQUENCES

Using the Validator, you can record one or more *test sequence*s to a sequence file. The sequence file can be used as a source of reference in future simulation sessions, for example after changes in the 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.

In addition to test sequences you can play *event sequences*—a subset of test sequences—plain text files that specify a sequence of events and assignments. However, events can not be recorded in the same way as a test sequence. See *Event sequence files description*, page 355.

Output types

These types of output can be generated:

States	The entire state configuration for the system to which the command was given.
Action functions	The action function executed during a **Send Signal** or a **Send Event** command.
Signals	The entire queue after a **Send Signal** or a **Send Event** command.

Variables	The variables that have been assigned a new value during a **Send Signal** or a **Send Event** command (not necessarily another value, but an assignment that 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 353.

BRIEFLY ABOUT PLAYING RECORDED TEST SEQUENCES

You can play recorded test sequences from the sequence file. This allows you to check if two different simulations give the same result, for example after changing the design model. 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 when debugging the model using RealLink.

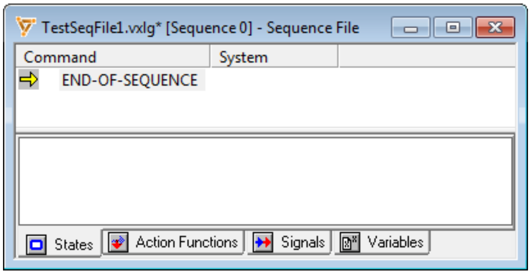
Recording and playing your test sequences

What do you want to do?

RECORDING A TEST SEQUENCE TO A SEQUENCE FILE

1 In the Validator, open your Validator workspace.

2 Choose **File>Sequence File>New** to open the **Sequence File** window.



3 Click the **Start recording** button ( ) on the **Debug** toolbar or choose **Debug>Record**.

4 Initialize the loaded system and send the Visual State 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 system, an error is issued.

5 Apply commands to the system. These are the commands that can be given and recorded in a test sequence file:

Initialize a system	Click the **Initialize System** button ( ) on the **Debug** toolbar (not available in target mode).
Send an event	Double-click an event in the **Event** window.
Set the values of internal and external variables, and action return values	Choose **Set Value** from the context menu in the **Variable** window (values of action return values are not available in target mode). See also *Changing values of variables*, page 332.
Force the system into a specific state	Choose **Force State** from the context menu in the **Systems** window (not available in target mode).
Send a signal to the system	In the **Signal Queues** window, double-click a signal.

The commands applied to the state machine model will be recorded and appended to the selected sequence.

If manual (interactive) simulation is performed on multiple systems, global events are sent to all systems and will be recorded once for each system that receives the event. This way of recording ensures that it is possible to repeat the test sequence by playing it.

Note: If you are recording a test sequence, *all* commands will be recorded, both manually applied commands and commands from a recorded test sequence file.

6 To stop recording, click the **Stop / Reset** button (■). The test sequence is saved automatically.

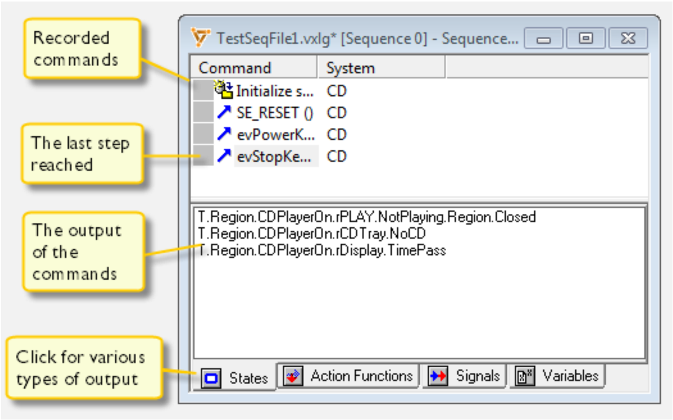
To find various commands for recording, right-click in the window to open the context menu. See also *Sequence File window*, page 383.

VIEWING OUTPUT FROM STEPS

The output from steps (commands) recorded to a sequence file during simulation can be viewed by selecting the appropriate command (step) in the **Sequence File** window.

1 In the Validator, choose **File>Sequence File>Open** to open the **Sequence File** window.

2 Right-click in the window and choose **Step Results** to display the output pane in the window.



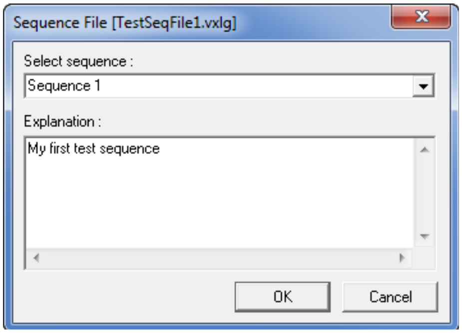
PLAYING YOUR RECORDED TEST SEQUENCE

1 In the Validator, open your workspace.

2 Choose **File>Sequence File>Open** and specify the file to use. The **Sequence File** window is displayed.

If the output pane is not already displayed, right-click in the window and choose **Step results** from the context menu.

3 Right-click in the window, and choose **Sequence>Select Sequence** from the context menu. The **Sequence File** dialog box is displayed:



Choose the test sequence to use and click **OK**.

4 To execute the steps in the test sequence one by one, click the **Step forward** button ( ) on the **Debug** toolbar.

5 To play the recorded test sequence automatically, click the **Play** button ( ) on the **Debug** toolbar.

The default speed is **Free Run**, which is the highest possible speed of the host computer. To change the speed, choose **Edit>Speed** and select the appropriate speed.

6 If you know exactly on which step to break execution, either select the step and choose **Play to cursor** from the context menu. Or, set a *stop point* on the specific step by double-clicking the step.

7 To search for some specific conditions to be fulfilled, use breakpoints which also work for commands sent from a recorded test sequence. See *Using breakpoints*, page 330.

8 To pause the execution, click the **Pause** button ( ) on the **Debug** toolbar.

9 To stop the 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

Jumping around in the recorded sequence is particularly useful if the signal queue in a recorded test sequence does not correspond to the one generated at runtime.

1 To jump around in the recorded test sequence, right-click in the **Sequence File** window and choose **Set as Next Step** from the context menu.

Execution of the recorded test sequence will stop if the sequence tries to send a signal that is different from the first signal in the queue. To continue execution in such a situation, continue with step 2.

2 Open the sequence file and choose the sequence.

3 As the next command to be executed, select the first command that is not a signal in the **Sequence File** window.

4 To manually empty the existing queue, click the **Empty Signal Queues** button on the **Debug** toolbar (or use the same command from the context menu).

You can now continue to play the test sequence.

COMPARING PLAYED TEST SEQUENCES WITH RECORDED OUTPUT

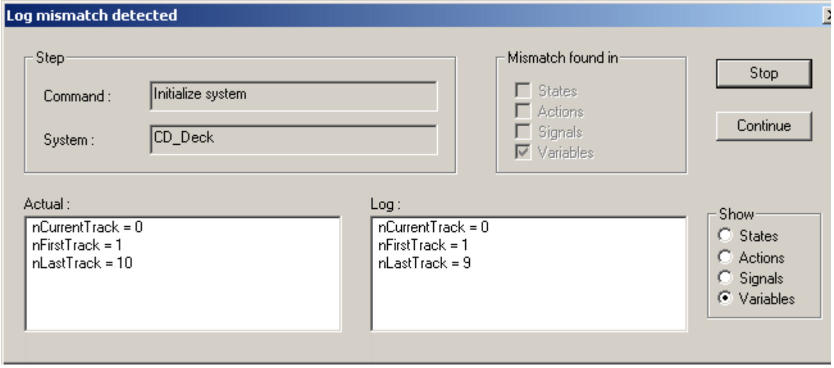
1 In the Validator, open your workspace.

2 Choose **File>Sequence File>Open** and specify the file to use. Right-click in the window and choose **Check>All** from the context menu.

If the output pane is not already displayed, right-click in the window and choose **Step results** from the context menu. Then right-click in the **Output** pane, choose **Check** and select the items you want to compare.

3 Play the sequence, see *Playing your recorded test sequence*, page 352.

If a design change has been made that results in a mismatch, execution will stop when you play the recorded test sequence. The Validator will report the mismatches caused by the change:



Event sequence files description

Event sequence files have the filename extension `vesq`.

SYNTAX

Event sequence files must conform to the following syntax, where terminals are set in single quotes ('):

```
entSequenceFile ::= 'SYSTEM' <identifier> <index> ( 'INITRESET' |
                    'NOINIT' ) [ FunctionReturns ] Steps
FunctionReturns ::= FunctionReturn [ FunctionReturns ]
FunctionReturn ::= 'FUNCRET' <identifier> Values
Values ::= Value [ ',' Values ]
Value ::= Constant
Steps ::= Step [ Steps ]
Step ::= Event | Assignment
Event ::= <identifier> '(' [ Parameters ] ')'
Parameters ::= Parameter [ ',' Parameters ]
Parameter ::= Constant
Assignment ::= InternalAssignment | ExternalAssignment
InternalAssignment ::= 'INTERNAL' <identifier>
                       [ '[' <index> ']' ] '=' Constant
ExternalAssignment ::= <identifier> '=' Constant
Constant ::= <int constant> | <float constant> | <hex constant> |
             <char constant>
```

The header consists of a system identifier and an index, and designates the system instance that the event sequence shall apply to. If there is only one instance of the system, the index must be `0`.

The keyword alternative consisting of `INITRESET` and `NOINIT` designates two different variants of event sequence files:

- `INITRESET` starts the event sequence at the initial state of the model loaded into the Validator, in other words with initializing the model followed by a reset event.
- `NOINIT` starts the event sequence at whatever state the model loaded into the Validator currently is in.

Event sequence files accept C-style comments.

Example of an event sequence file

```
/*
   Example Event Sequence File
*/

SYSTEM System1 0

INITRESET

FUNCRET Action1 1,2,3,4
FUNCRET Action2 1.0

Event1()
External1 = 1
Event2(1)
INTERNAL Internal1 = 0xF
Event3(1.0)
INTERNAL Internal2[2] = 'c'
Event4(0x1)
Event5('a')
Event6(1, 1.0, 0x1, 'a')

/* End of Example Event Sequence File */
```

The Visual State Validator

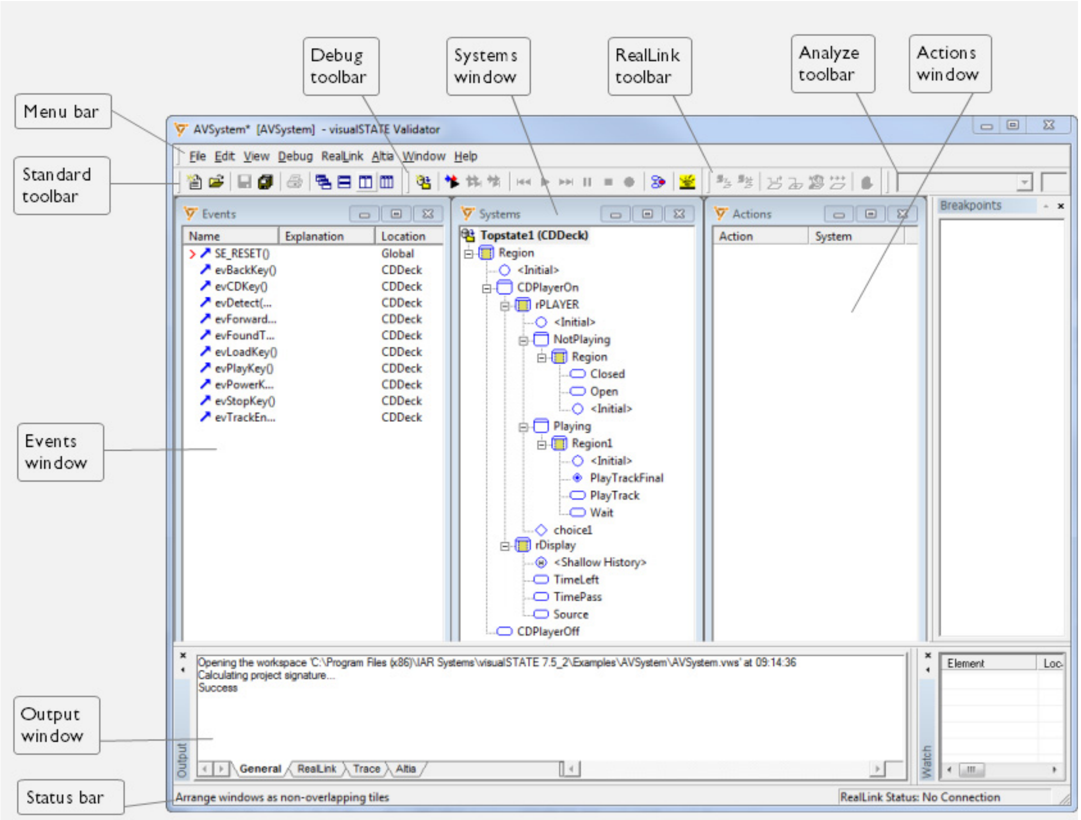
Introduction to the Visual State Validator

Learn more about:

- *Briefly about the Visual State Validator*, page 358

BRIEFLY ABOUT THE VISUAL STATE VALIDATOR

The Validator has a number of windows that provide information during validation. All windows have context menus where you can activate various commands. The Validator windows are opened via the **Windows** menu and the **View** menu.



The Validator workspace contains information on your validation session (filename extension `vws`). The file contains 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.

You can have more than one Validator workspace, each loading the same project, and each having its own particular setup. This is useful when testing different aspects of your state machine model. Note that it is only possible to have one project in a workspace.

When you start the Validator from the Navigator you will automatically get an appropriate workspace for the project in the Validator.

Graphical environment for the Validator

Reference information about:

- *The Validator main window*, page 360
- *Actions window*, page 364
- *Animation Speed dialog box*, page 365
- *Breakpoint Reached dialog box*, page 365
- *Breakpoints window*, page 366
- *Breakpoints Setup dialog box*, page 367
- *Customize Graphical Animation dialog box*, page 338
- *Dynamic Analysis window*, page 375
- *Events window*, page 377
- *Find Trace dialog box*, page 379
- *Guard Expressions window*, page 380
- *Log Mismatch Detected dialog box*, page 381
- *Output window*, page 382
- *Sequence File window*, page 383
- *Sequence File dialog box*, page 386
- *Set Event Parameter Value dialog box*, page 386
- *Signal Queues window*, page 387
- *Static Analysis window*, page 388
- *Systems window*, page 389
- *System Setup window*, page 391
- *Timer Tick Length dialog box*, page 392
- *Timers window*, page 392
- *Trace Point Setup dialog box*, page 394
- *Variables window*, page 395
- *Watch window*, page 398

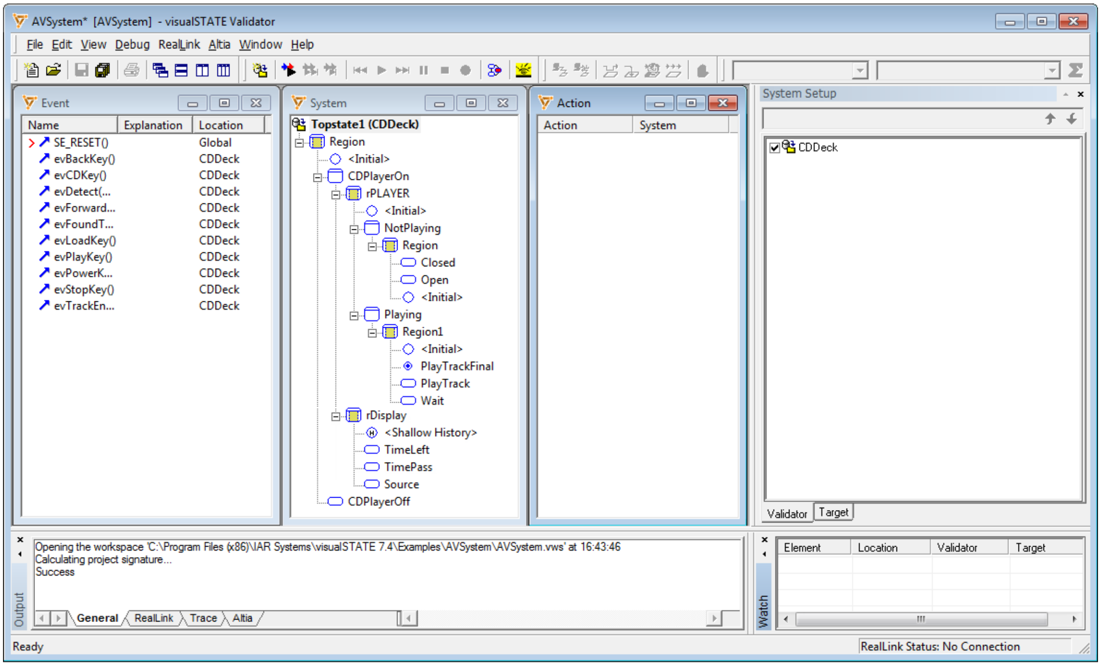
See also:

- *Open Altia Model dialog box*, page 906
- *Connect Elements dialog box*, page 903

- *Define Altia Properties dialog box*, page 905
- *RealLink Options dialog box*, page 809
- *RealLink Properties dialog box*, page 806
- *RealLink RS232 Communication Setup dialog box*, page 808
- *RealLink TCP/IP Communication Setup dialog box*, page 807.

The Validator main window

The main window of the Validator is displayed when you start the Validator.



The screenshot shows the window and its default layout.

The main window is a container for the various Validator windows.

Menu bar

The menu bar contains:

File

Commands for creating, opening, and saving workspaces, sequence files, and analyses, loading projects, printing, and exiting the Validator. See *File menu*, page 400.

Edit

Commands for undoing recent actions, making settings, and setting up breakpoints. See *Edit menu*, page 401.

View

Commands for opening windows and controlling which toolbars to display. See *View menu*, page 403.

Debug

Commands for simulating your model. See *Debug menu*, page 404.

RealLink

Commands for debugging your model in a target application using RealLink. See *RealLink menu*, page 804.

Altia

Commands for prototyping and simulating a graphical interface of your model, using Altia Design. See *Prototyping a graphical interface*, page 883.

Window

Commands for changing how the Validator windows are arranged on the screen. See *Window menu*, page 406.

Help

Commands that provide information about the Validator. See *Help menu*, page 407.

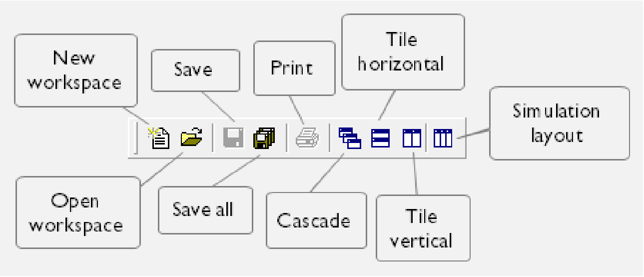
For more information about each menu, see *Reference information on Validator menus*, page 399.

Standard toolbar

The standard toolbar—available from the **View** menu—provides buttons for the most frequently used commands on the Validator menus.

For a description of a button, point to it with the mouse pointer. The name of the button is displayed as a tooltip and a description is displayed in the status bar at the bottom of the main window. When a command is not available, the corresponding toolbar button is dimmed, and you will not be able to click it.

This figure shows the menu commands that correspond to each of the toolbar buttons:



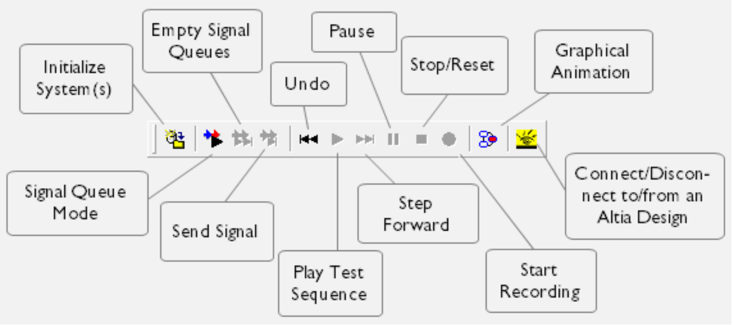
See *File menu*, page 400.

Debug toolbar

The Debug toolbar—available from the **View** menu—provides buttons for simulating the execution of your model using the Validator.

For a description of a button, point to it with the mouse pointer. The name of the button is displayed as a tooltip and a description is displayed in the status bar at the bottom of the main window. When a command is not available, the corresponding toolbar button is dimmed, and you will not be able to click it.

This figure shows the menu commands that correspond to each of the toolbar buttons:



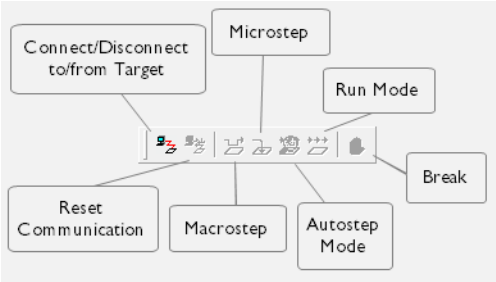
See *Debug menu*, page 404.

RealLink toolbar

The RealLink toolbar—available from the **View** menu—provides buttons for debugging the execution of your model using RealLink.

For a description of a button, point to it with the mouse pointer. The name of the button is displayed as a tooltip and a description is displayed in the status bar at the bottom of the main window. When a command is not available, the corresponding toolbar button is dimmed, and you will not be able to click it.

This figure shows the menu commands that correspond to each of the toolbar buttons:



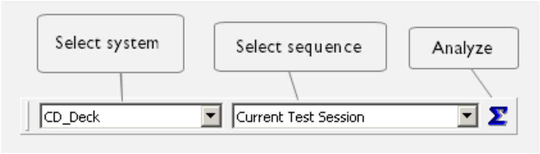
See *RealLink menu*, page 804.

Analyze toolbar

The Analyze toolbar—available from the **View** menu—provides buttons for analyzing your model.

For a description of a button, point to it with the mouse pointer. The name of the button is displayed as a tooltip and a description is displayed in the status bar at the bottom of the main window. When a command is not available, the corresponding toolbar button is dimmed, and you will not be able to click it.

This figure shows the menu commands that correspond to each of the toolbar buttons:

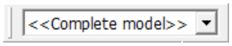


See *Analyzing*, page 345.

Variant toolbar

The Variant toolbar—available from the **View** menu—controls the use of product variants in the model.

This figure shows the toolbar:



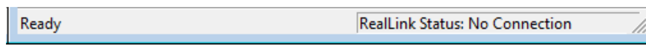
Variant selector

Choose which product *variant* that the Validator operates on. If you choose **<<Complete model>>**, the Validator will operate on the entire model. The feature sets of the variants are edited inside the Designer.

If you change the variant, the Validator will reload the model.

Status bar

The status bar at the bottom of the window can be enabled from the **View** menu.

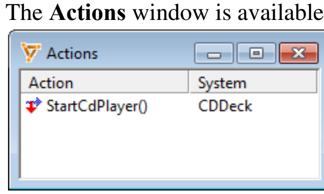


The status bar displays:

- Descriptions of menu commands when you open a menu and hover over commands
- Descriptions of toolbar buttons that you hover over with the mouse pointer
- The status of processes in the Validator.

Actions window

The **Actions** window is available from the **Window>New Window** submenu.



This window contains information about the most recent deduction. The display area shows values assigned to the variables, executed actions, and the arguments with which the actions were called.

Context menu

This context menu is available:

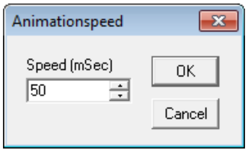
This command is available:

Show target values

Toggles the display mode between showing a representation of the state machine model as it appears in Validator mode and in Target mode. Requires that the Validator is connected to a target system via RealLink.

Animation Speed dialog box

The **Animation Speed** dialog box is available from the **Edit>Speed** submenu.



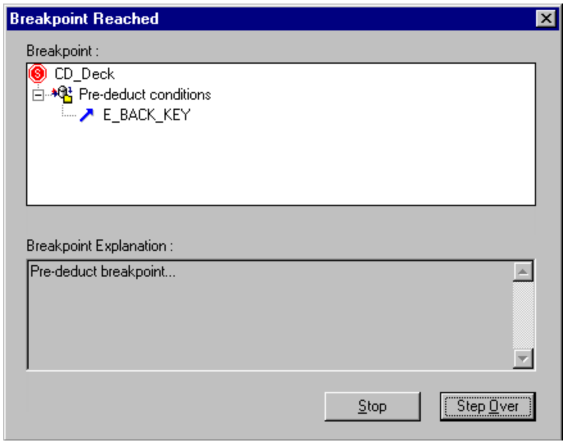
Use this dialog box to set a specific execution speed for test sequence files.

Speed

Specify the number of milliseconds to pass between each step in the animation.

Breakpoint Reached dialog box

The **Breakpoint Reached** dialog box is displayed when a breakpoint is reached.



This dialog box displays the breakpoint that has been reached and lets you decide how to proceed. See *Using breakpoints*, page 330.

Breakpoints

Displays the breakpoint that was reached, in the same format as in the **Breakpoints** window.

Breakpoint Explanation

Displays the description you gave the breakpoint when you defined it.

Stop

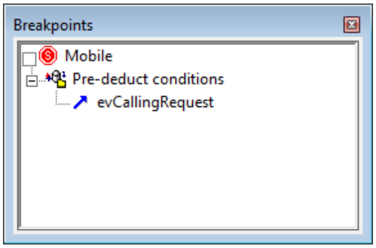
Stops the simulation.

Step Over

Steps over the breakpoint and performs the deduction.

Breakpoints window

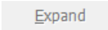
The **Breakpoints** window is available from the **View** menu.



This window displays all defined breakpoints. Use the checkbox to enable or disable them. See *Defining breakpoints*, page 329.

Context menu

This context menu is available:



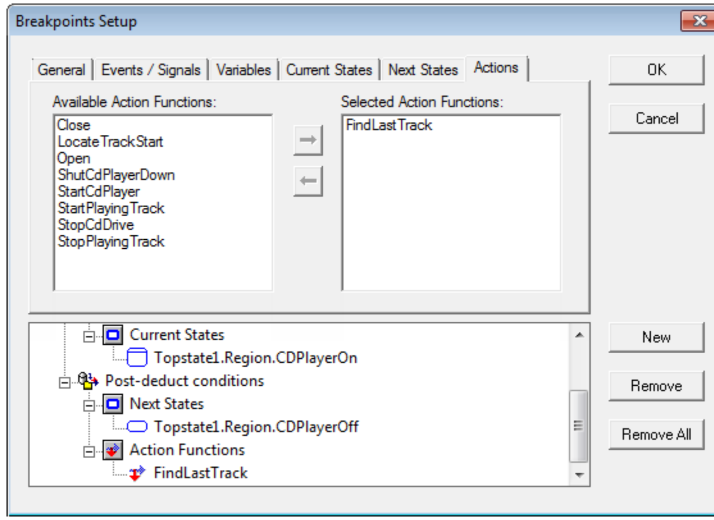
This command is available:

Expand

Shows the detailed breakpoint conditions.

Breakpoints Setup dialog box

The **Breakpoints Setup** dialog box is available from the **Edit** menu.



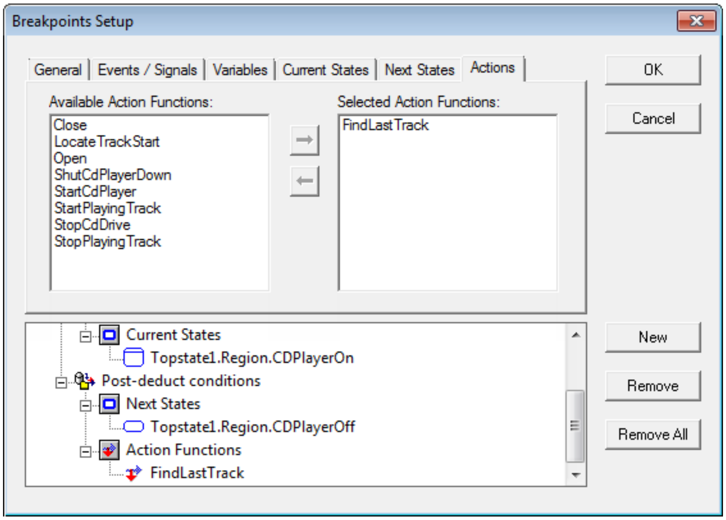
Use this dialog box to set breakpoint options for the Validator.

You can set options on these tabbed pages:

- *Breakpoints Setup dialog box : Actions*, page 368
- *Breakpoints Setup dialog box : Current States*, page 369
- *Breakpoints Setup dialog box : Events/Signals*, page 370
- *Breakpoints Setup dialog box : General*, page 371
- *Breakpoints Setup dialog box : Next States*, page 372
- *Breakpoints Setup dialog box : Variables*, page 373

Breakpoints Setup dialog box : Actions

The **Breakpoints Setup** dialog box is available from the **Edit** menu.



Use the **Actions** page to connect an action function to a breakpoint. See *Defining breakpoints*, page 329.

Available action functions

Displays the available action functions that can be connected to the selected breakpoint in the display area. Double-click an action function to connect it to the breakpoint, or select it and click the **Right Arrow** button.

Selected action functions

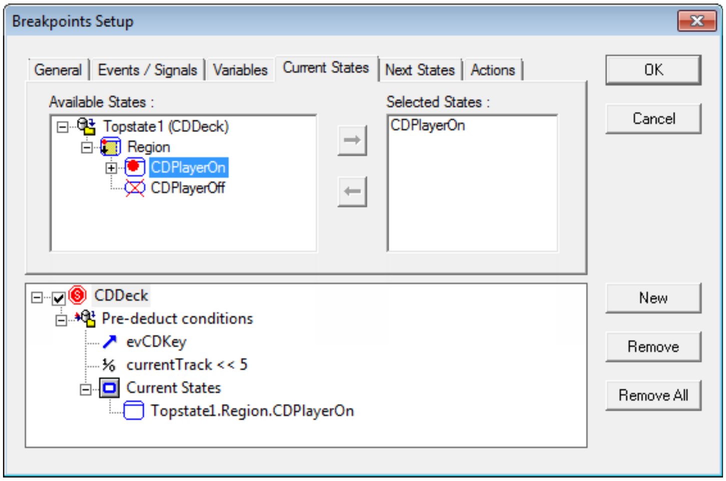
Displays the action functions that have been connected to the selected breakpoint in the display area. Double-click an action function to remove it from the breakpoint, or select it and click the **Left Arrow** button.

Display area and buttons

See *Breakpoints Setup dialog box : General*, page 371.

Breakpoints Setup dialog box : Current States

The **Breakpoints Setup** dialog box is available from the **Edit** menu.



Use the **Current States** page to specify a specific state as a breakpoint condition. The condition will be evaluated *before* a deduction is performed. See *Defining breakpoints*, page 329.

Available states

Displays the available states that can be used as a condition for the selected breakpoint in the display area. Double-click a state to connect it to the breakpoint, or select it and click the **Right Arrow** button →.

Selected states

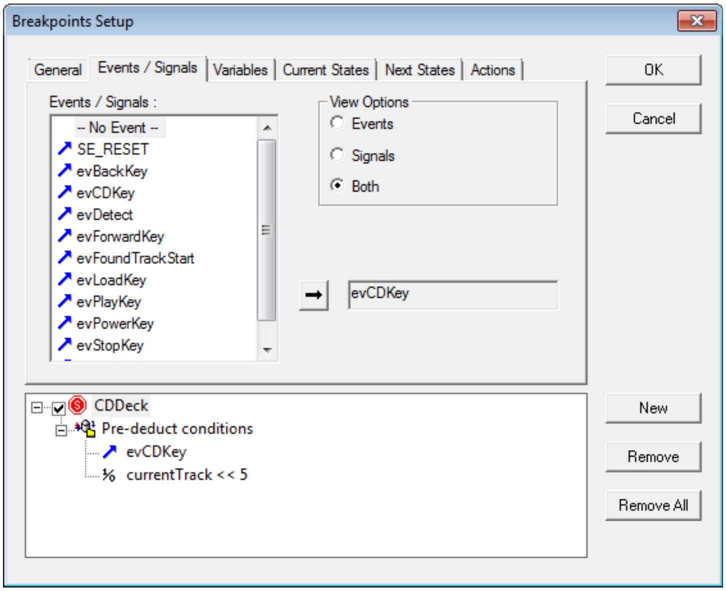
Displays the states that are used as conditions for the selected breakpoint in the display area. Double-click a state to remove it from the breakpoint, or select it and click the **Left Arrow** button ←.

Display area and buttons

See *Breakpoints Setup dialog box : General*, page 371.

Breakpoints Setup dialog box : Events/Signals

The **Breakpoints Setup** dialog box is available from the **Edit** menu.



Use the **Events/Signals** page to specify a specific event or signal as a breakpoint condition. See *Defining breakpoints*, page 329.

Events/signals

Displays the available events and signals that can be used as a condition for the selected breakpoint in the display area. Double-click an event or a signal to connect it to the breakpoint, or select it and click the **Right Arrow** button ➡.

View options

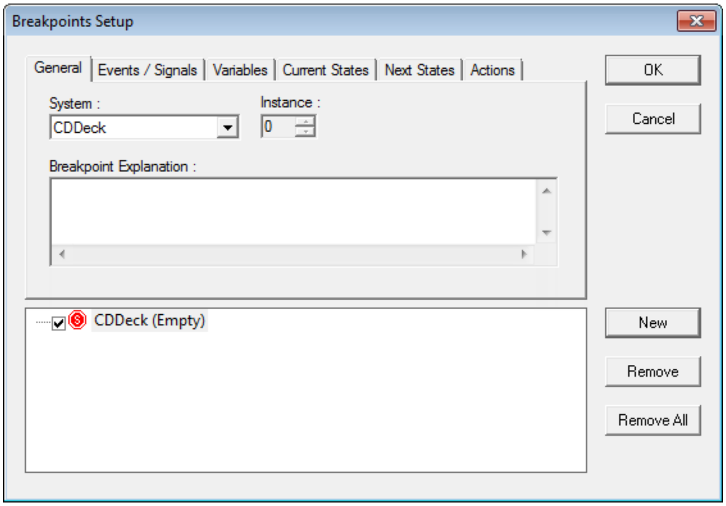
Choose what to show in the **Events/Signals** list: **Events**, **Signals**, or **Both**.

Display area and buttons

See *Breakpoints Setup dialog box : General*, page 371.

Breakpoints Setup dialog box : General

The **Breakpoints Setup** dialog box is available from the **Edit** menu.



Use the **General** page to create a breakpoint and make basic settings for it. See *Defining breakpoints*, page 329.

System

Choose the system that the breakpoint should be applied to.

Instance

If there are more than one instance of the selected system, choose which one to apply the breakpoint to. See *Reuse of design using system instances*, page 126.

Breakpoint explanation

Type a description of the breakpoint.

Display area

Displays all defined breakpoints. Select a breakpoint to enable it.

New

Creates a new breakpoint for the selected system.

Remove

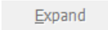
Removes the selected breakpoint.

Remove All

Removes all breakpoints.

Context menu

This context menu is available in the display area:



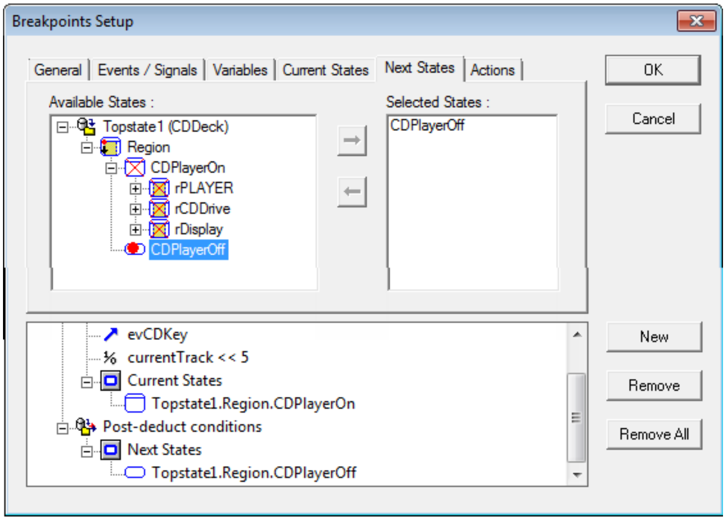
This command is available:

Expand

Shows the detailed breakpoint conditions.

Breakpoints Setup dialog box : Next States

The **Breakpoints Setup** dialog box is available from the **Edit** menu.



Use the **Next States** page to specify a specific state as a breakpoint condition. The condition will be evaluated *after* a deduction is performed. See *Defining breakpoints*, page 329.

Available states

Displays the available states that can be used as a condition for the selected breakpoint in the display area. Double-click a state to connect it to the breakpoint, or select it and click the **Right Arrow** button .

Selected states

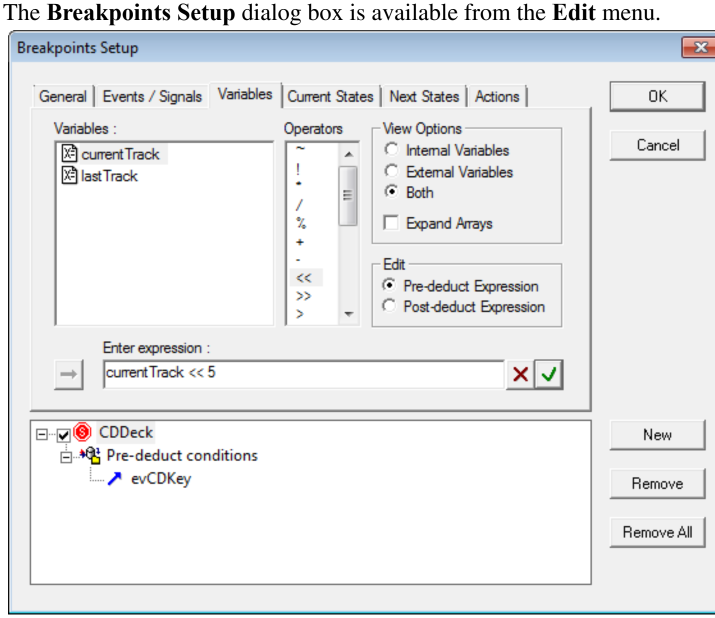
Displays the states that are used as conditions for the selected breakpoint in the display area. Double-click a state to remove it from the breakpoint, or select it and click the **Left Arrow** button.

Display area and buttons

See *Breakpoints Setup dialog box : General*, page 371.

Breakpoints Setup dialog box : Variables

The **Breakpoints Setup** dialog box is available from the **Edit** menu.



Use the **Variables** page to specify an expression as a breakpoint condition. See *Defining breakpoints*, page 329.

Variables

Lists the available variables for use in a guard expression used as a breakpoint condition. Double-click a variable to use it in an expression, or select it and click the **Right Arrow** button.

Operators

Lists the available operators for use in a guard expression used as a breakpoint condition. Double-click an operator to use it in an expression, or select it and click the **Right Arrow** button .

View options

Choose what to show in the **Variables** list: **External variables**, **Internal variables**, or **Both**.

Expand arrays

Displays arrays expanded with all members visible.

Edit

Choose whether to evaluate the guard expression before or after the deduction.

Enter expression

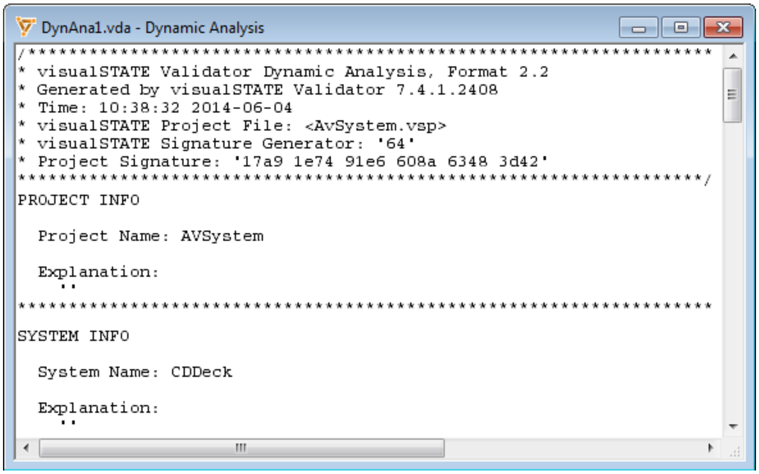
Compose the guard expression to be used as a breakpoint condition by typing and by using the **Variables** and the **Operators** lists. Click the **Apply** button to apply the expression or the **Clear** button to clear the field without applying the expression.

Display area and buttons

See *Breakpoints Setup dialog box : General*, page 371.

Dynamic Analysis window

The **Dynamic Analysis** window is available from the **File>Analysis** submenu.



This window contains the results of a dynamic analysis of your model when you click the **Analyze** button on the **Analysis** toolbar or choose **Debug>Analyze**.

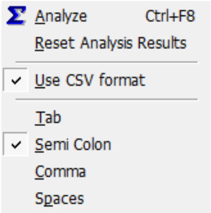
The text consists of two sections:

- The summary section shows the calculated coverage percentage and the most frequently activated elements of those covered by the analysis.
- The details section shows how many times a specific element has been activated, and frequency calculated as a percentage of the entire activation of this group of identifiers.

See also *Analyzing*, page 345.

Context menu

This context menu is available:



These commands are available:

Analyze

Analyzes the system selected in the **Analysis** toolbar, using the selected test sequence.

Reset Analysis Results

Resets the analysis results.

Use CSV Format

Formats the contents of the window in the CSV (comma-separated value) format, using one of the delimiters below.

Tab

Uses tabs as CSV delimiters.

Semicolon

Uses semicolons as CSV delimiters.

Comma

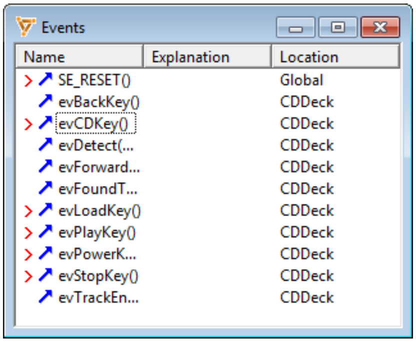
Uses commas as CSV delimiters.

Spaces

Uses space characters as CSV delimiters.

Events window

The **Events** window is available from the **Window>New Window** submenu.



This window provides a view of all events defined in the loaded project, and is used for sending events into the system(s).

See also:

- *Events*, page 179
- *Sending events manually*, page 327.

Name

The name of an event. Active events (events that will trigger transitions if sent) have a red > mark to the left.

Explanation

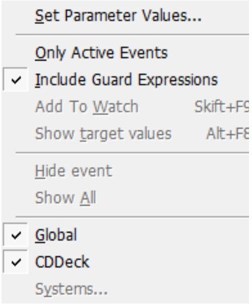
The description you have given the event (if any).

Location

The location of the event definition.

Context menu

This context menu is available:



These commands are available:

Set Parameter Values

Displays the **Set Event Parameter Value** dialog box, see *Set Event Parameter Value dialog box*, page 386.

Only Active Events

Shows/hides events that are not active.

Include Guard Expressions

Enables/disables guard expressions as a factor when determining whether an event is active.

Add to Watch

Adds the selected event to the **Watch** window, see *Watch window*, page 398.

Show target values

Toggles the display mode between showing a representation of the state machine model in Validator mode and in the target mode. Requires that the Validator is connected to a target system via RealLink.

Hide Event

Hides the selected event. To show the event again, choose **Show All** from the context menu.

Show All

Shows events that have been hidden using the **Hide Event** command.

Global

Shows/hides globally defined events.

System

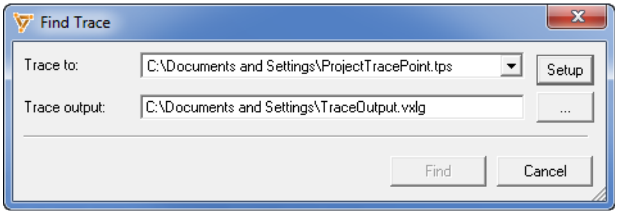
Shows/hides events defined locally in the named system.

Systems

If there is more than one system, chooses one of the system.

Find Trace dialog box

The **Find Trace** dialog box is available from the **Debug** menu.



Use this dialog box to set up a trace, a sequence of steps that leads to a desired configuration of states.

See also *Tracing using the Validator*, page 341.

Trace to

Specify the configuration to reach. Choose between:

<Initial>

Performs a trace to the initial state in the system.

<Current>

Performs a trace to the current state in the system.

<Specify file>

Makes the **Setup** button available. This button opens the **Trace Point Setup** dialog box.

Trace output

Specify the name of the file to save the resulting test sequence file to, or browse to an existing output file using the browse button.

Setup

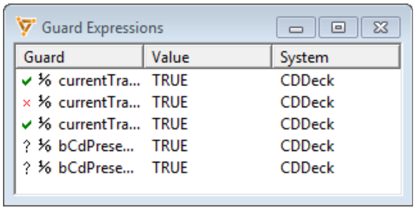
Displays the **Trace Point Setup** dialog box, see *Trace Point Setup dialog box*, page 394.

Find

Starts the trace. The Validator will by means of the Verificator try to find a trace to the specified state configuration. The resulting sequence file will be saved, if a trace can be found.

Guard Expressions window

The **Guard Expressions** window is available from the **Window>New Window** submenu.



This window displays all guard expressions defined in all systems.

Guard

The name of the expression. The characters to the left indicate:

Green check mark

The expression has been evaluated to true.

Red cross

The expression has been evaluated to false.

Question mark

The expression cannot be evaluated.

Value

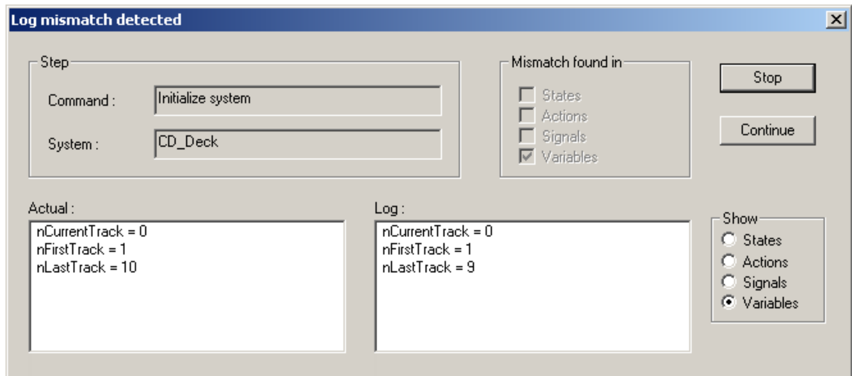
A boolean value that reflects whether the expression currently evaluates to true or false, if that is known.

System

The name of the system where the expression is defined.

Log Mismatch Detected dialog box

The **Log Mismatch Detected** dialog box is displayed when a recorded test sequence is played and mismatches in output are detected.



This dialog box contains reports of mismatches detected when a recorded test sequence is played, caused by design changes.

See *Recording and playing test/event sequences*, page 349.

Command

The command given in the current step, see *Sequence File window*, page 383.

System

The name of the system that the command was applied to.

Mismatch found in

Shows in which type of output the mismatch was found.

Stop

Stops playing the test sequence.

Continue

Continues to play the test sequence.

Actual

Displays the output from the current playing if the test sequence.

Log

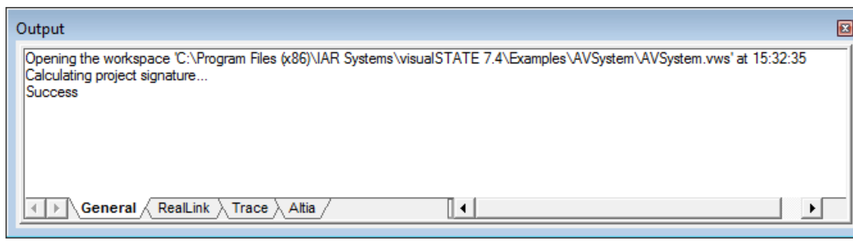
Displays the logged output from the previously recorded test sequence.

Show

Choose the type of output you want to see mismatches for.

Output window

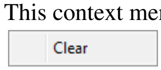
The **Output** window is available from the **View** menu.



This window displays information about the loaded workspace. The tabbed pages contain general information from the Validator, RealLink, and Altia when these tools are running, as well as trace information.

Context menu

This context menu is available:



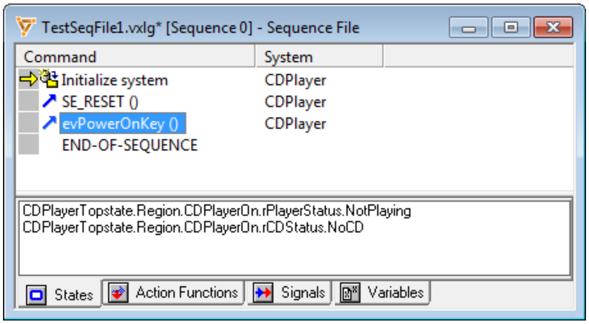
This command is available:

Clear

Deletes all text for the active view in the window.

Sequence File window

The **Sequence File** window is available from the **File>Sequence File** submenu.



Use this window to record a test sequence to a sequence file. A test sequence consists of a number of steps that each describes a command and the output produced by it.

See also *Recording and playing test/event sequences*, page 349.

Command

Displays the commands that have been added to the sequence.

System

Displays the system that each command was applied to.

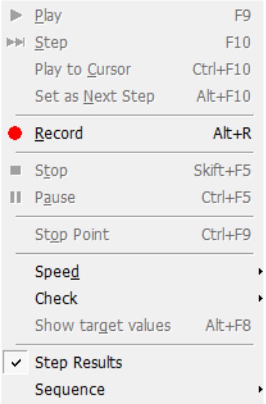
Output area

Displays the results of the selected command in the **Commands** column, for each of the four types of output: states, action functions, signals, and variables. Click the tab for the type of output you want to see.

If this area is not visible, right-click in the window and choose **Step Results** from the context menu.

Context menu

This context menu is available:



These commands are available:

Play

Plays the test sequence.

Step

Plays the test sequence one step forward.

Play to Cursor

Plays the test sequence up to the selected command.

Set as Next Step

Sets the selected command as the next step to be processed.

Record

Starts recording a test sequence.

Stop

Stops playing or recording the test sequence and resets it.

Pause

Pauses the test sequence.

Stop Point

Stops playing the test sequence at the selected command (stop point).

Speed

Opens a submenu where you can set the time to pass between each step in the animation.

Speed>Free Run

Plays the animation as fast as possible.

Speed>Define

Opens the **Animation Speed** dialog box, see *Animation Speed dialog box*, page 365.

Check

Opens a submenu where you choose which types of output to validate during the animation.

Show target values

Toggles the display mode between showing a representation of the state machine model in Validator mode or in Target mode. Requires that the Validator is connected to a target system via RealLink.

Step Results

Shows/hides the output area of the window.

Sequence>Select Sequence

Opens the **Sequence File** dialog box, see *Sequence File dialog box*, page 386.

Sequence>Next Sequence

Makes the next sequence recorded in the file active.

Sequence>Previous Sequence

Makes the previous sequence recorded in the file active.

Sequence>New Sequence

Creates a new sequence in the file.

Sequence>Reset Sequence

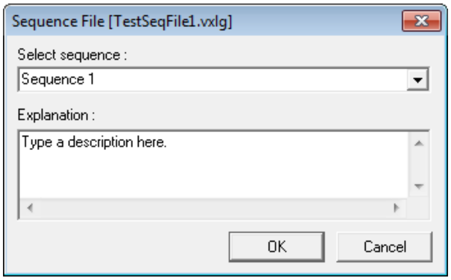
Resets the current sequence by removing all commands.

Sequence>Delete Sequence

Deletes the current sequence entirely.

Sequence File dialog box

The **Sequence File** dialog box is available from the context menu in the **Sequence File** window.



Use this dialog box to give a sequence a name and a description, and to save it.

See also *Recording and playing test/event sequences*, page 349.

Select Sequence

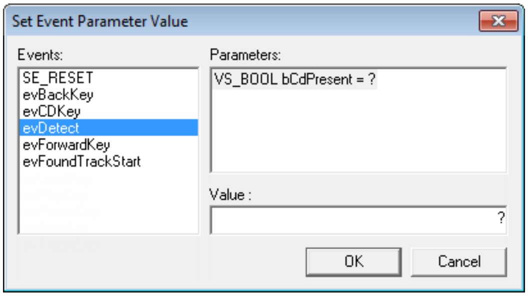
Type a name for the sequence.

Explanation

Type a description for the sequence.

Set Event Parameter Value dialog box

The **Set Event Parameter Value** dialog box is available from the **Events** window context menu.



Use this dialog box to assign values to event parameters.

See also *Specifying event parameters*, page 326.

Events

Displays all events defined in the loaded project. Select the event you want to set parameters for.

Parameters

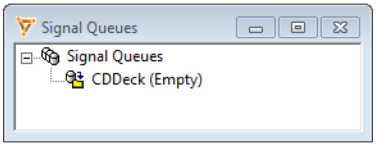
Displays all parameters of the selected event. Select the parameter you want to assign a value.

Value

Type a value for the selected parameter.

Signal Queues window

The **Signal Queue**s window is available from the **Window>New Window** submenu, and from the **Visual State** menu in the IAR Embedded Workbench IDE.

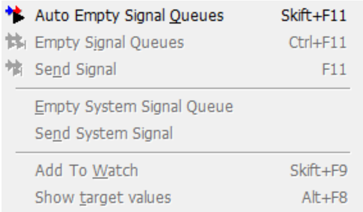


This window provides a view of the signal queues in all systems and instances. Use it for signal handling.

See also *Handling signal queues for a single system*, page 329.

Context menu

This context menu is available:



These commands are available:

Auto Empty Signal Queues

Enables/disables automatic emptying of signal queues. When the command is enabled, the signal queue is automatically emptied when an event is sent manually to a system. During execution, the signal queue is not emptied automatically.

Empty Signal Queues

Sends all signals in all queues one signal at a time, beginning with the first queue, until all queues are empty.

Send Signal

Sends the first signal in the first queue that contains signals. Arrange the order of the queues in the **System Setup** window, see *System Setup window*, page 391.

Empty System Signal Queue

Empties the signal queue for the selected system.

Send System Signal

Sends the first signal in the selected system.

Add to Watch

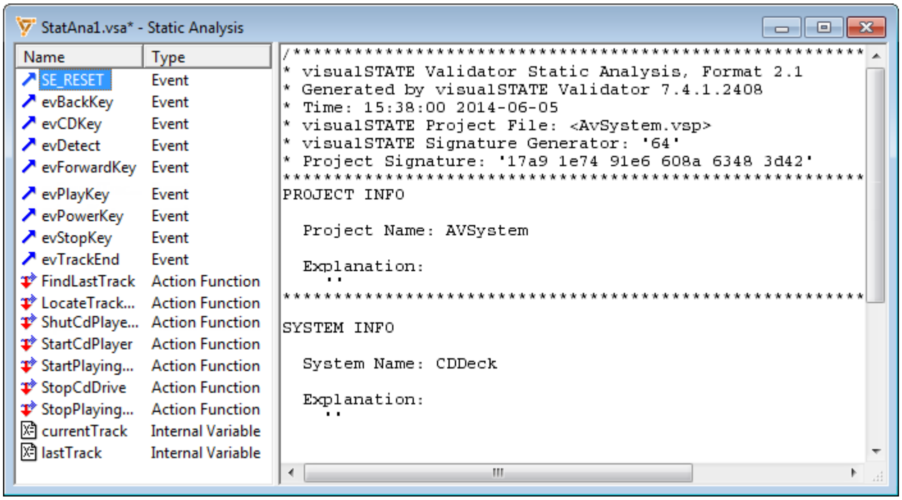
Adds the signal to the **Watch** window, see *Watch window*, page 398.

Show target values

Toggles the display mode between showing a representation of the state machine model in Validator mode or in Target mode. Requires that the Validator is connected to a target system via RealLink.

Static Analysis window

The **Static Analysis** window is available from the **File>Analysis** submenu.



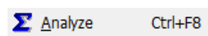
This window contains the results of a static analysis of the system selected in the **Analysis** toolbar. To perform an analysis, select one or more elements to analyze transitions for and click the **Analyze** button on the **Analysis** toolbar or choose **Debug>Analyze**.

The report gives an overview of the elements used in the transitions.

See also *Performing static analysis*, page 346.

Context menu

This context menu is available:



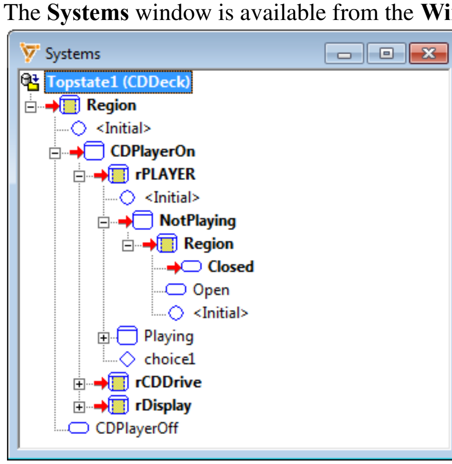
This command is available:

Analyze

Analyzes the selected events.

Systems window

The **Systems** window is available from the **Window>New Window** submenu.

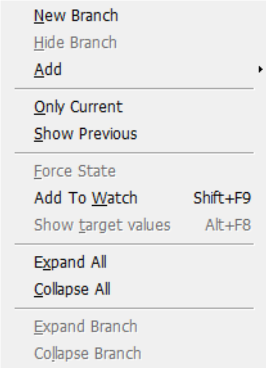


This window displays a hierarchical view of the systems in the project. The default view shows each system and each of their instances in a separate branch in the tree.

See also *Specifying the order of the systems/instances*, page 333.

Context menu

This context menu is available:



These commands are available:

New Branch

Adds the selected state as a new branch in the window.

Hide Branch

Hides the selected branch.

Add>*System*

Adds the system as a new branch in the window.

Add>New System Window

Opens a new instance of the **System** window with the same contents (initially) as the current one.

Only Current

Displays only the states that became current upon sending the most recent event.

Show Previous

Shows a visual indicator by the states that were current before the most recent event was sent.

Force State

Forces the system into the selected state. All states can be forced.

Add to Watch

Adds the selected state to the **Watch** window, see *Watch window*, page 398.

Show target value

Toggles the display mode between showing a representation of the state machine model in Validator mode or in the target mode. Requires that the Validator is connected to a target system via RealLink.

Expand All

Expands all branches in the window.

Collapse All

Collapses all branches in the window.

Expand Branch

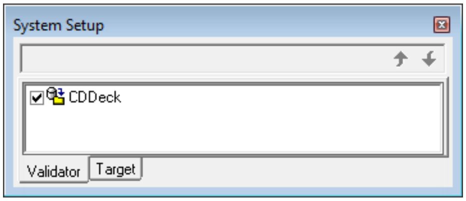
Expands the selected branch.

Collapse Branch

Collapses the selected branch.

System Setup window

The **System Setup** window is available from the **View** menu.



Use this window to set up the order in which the systems should be simulated. There is one tabbed page for the state machine model in Validator mode and one for the target mode, when the Validator is connected to a target system via RealLink.

Rearranging the order of the systems changes the order of how events are sent to them.

Note: The order of systems only applies to interactive simulation (simulation that does not use test sequence files).

See *Specifying the order of the systems/instances*, page 333.

Display area

Displays the systems in the project. Use the checkboxes to enable or disable systems. Disabled Systems will not receive events.

To rearrange the order of the project, use these buttons:

Move Up

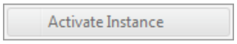
Moves the selected item upward in the list.

Move Down

Moves the selected item downward in the list.

Context menu

This context menu is available:



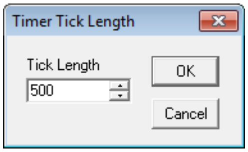
This command is available:

Activate Instance

Activates the selected instance.

Timer Tick Length dialog box

The **Timer Tick Length** dialog box is available from the **Edit>Timer Tick Length** submenu.



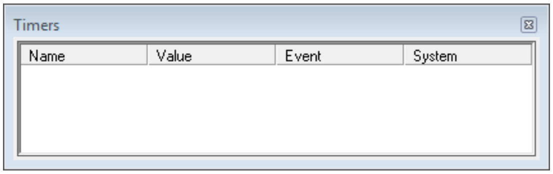
Use this dialog box to set the tick length of timer ticks used in the Validator.

Tick length

The timer tick length in milliseconds.

Timers window

The **Timers** window is available from the **View** menu.



This window displays the values of all running timers.

Name

The name of the timer.

Value

The current value of the timer.

Event

The event which the timer will send to the model when the timer times out.

System

The system where the timer is defined.

Context menu

This context menu is available:

These commands are available:

Stop Timer

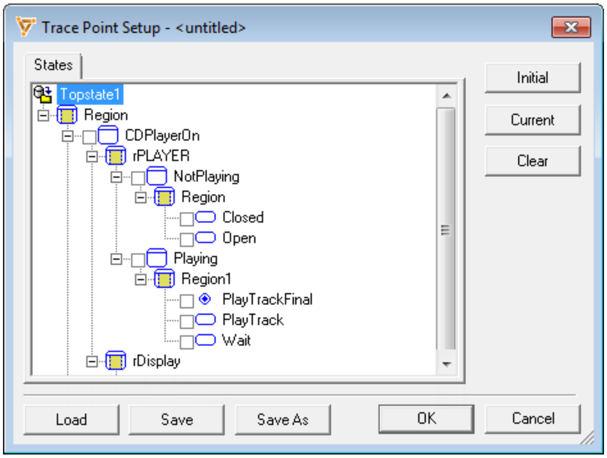
Stops the selected timer.

Stop All Timers

Stops all running timers.

Trace Point Setup dialog box

The **Trace Point Setup** dialog box is available from the **Trace Setup** dialog box.



This dialog box displays the states and regions of the system. Use it to create a trace point (the state configuration you want the trace to reach) by selecting the desired states, and save to a file.

See also *Setting up the trace point*, page 343.

States

Displays the states and regions of the system.

Initial

Selects the initial state(s) in the system as the trace point.

Current

Selects the current state(s) in the system as the trace point.

Clear

Clears the trace point.

Load

Displays a standard dialog box for navigating to an existing trace point setup file to load.

Save

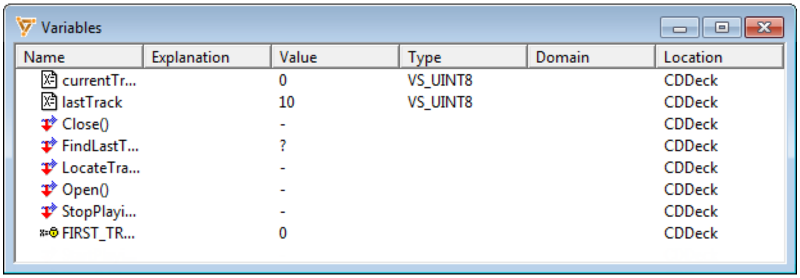
Displays a standard dialog box for saving the state configuration to a trace point setup file.

Save As

Displays a standard dialog box for saving the trace point setup file under another name.

Variables window

The **Variables** window is available from the **Window>New Window** submenu.



This window lists all variables, action functions, and constants declared in all systems.

See also *Changing values of variables*, page 332.

See also *Visual State operands, reference information*, page 196 and *Creating a transition element*, page 184.

Name

The name of the variable, action function, or constant.

Explanation

The description you have given the variable (if any).

To display this column, chose **View>Field Chooser**. Make sure the **Variable** window is the active window. Select **Explanation** in the **Field Chooser** window.

Value

The current value of the variable, action function, or constant. Click to edit it.

Type

The type of the internal variable, if any.

To display this column, chose **View>Field Chooser**. Make sure the **Variable** window is the active window. Select **Type** in the **Field Chooser** window.

Domain

The domain for an internal variable, if any.

To display this column, chose **View>Field Chooser**. Make sure the **Variable** window is the active window. Select **Domain** in the **Field Chooser** window.

Location

The system the variable is located in, or if the variable is global.

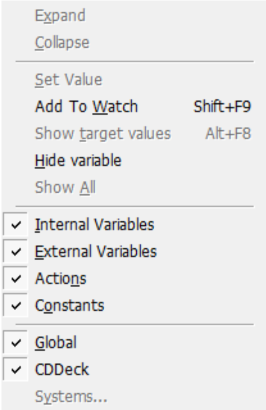
To display this column, chose **View>Field Chooser**. Make sure the **Variable** window is the active window. Select **Location** in the **Field Chooser** window.

Value

The current value of the variable, action function, or constant. Click to edit it.

Context menu

This context menu is available:



These commands are available:

Expand

Expands the item to show all values.

Collapse

Collapses the item to hide values.

Set Value

Makes the value field editable for the selected item.

Add to Watch

Adds the selected item to the **Watch** window, see *Watch window*, page 398.

Show target values

Toggles the display mode between showing a representation of the state machine model in Validator mode or in Target mode. Requires that the Validator is connected to a target system via RealLink.

Hide Variable

Hides the selected item. To show it again, choose **Show All** from the context menu.

Show All

Shows variables, action functions, and constants that have been hidden using the **Hide Variable** command.

Internal Variables

Shows/hides internal variables.

External Variables

Shows/hides external variables.

Actions

Shows/hides actions.

Constants

Shows/hides constants.

Global

Shows/hides all globally declared variables, action functions, and constants.

System

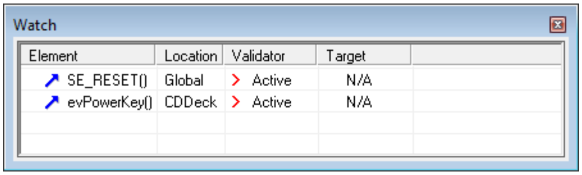
Shows/hides variables, action functions, and constants declared locally in the named system.

Systems

If there is more than one system, chooses one of the systems.

Watch window

The **Watch** window is available from the **View** menu.



This window contains a collection of elements that you might want to monitor, added from the **Systems**, **Events**, **Variables**, and **Signal Queues** windows.

Element

The name of the element that you are watching.

Location

The location of the element.

Validator

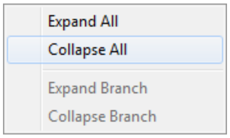
The status of the element in the design model.

Target

The status of the element as displayed in Target mode. Requires that the Validator is connected to a target system via RealLink.

Context menu

This context menu is available:



These commands are available:

Expand All

Expands all branches in the window.

Collapse All

Collapses all branches in the window.

Expand Branch

Expands the selected branch.

Collapse Branch

Collapses the selected branch.

Reference information on Validator menus

Reference information about:

- *File menu*, page 400
- *Edit menu*, page 401
- *View menu*, page 403
- *Debug menu*, page 404
- *Window menu*, page 406
- *Help menu*, page 407
- *Validator shortcut key summary*, page 407

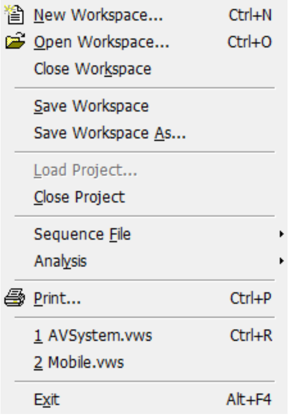
See also:

- *Visual State menu*, page 776
- *RealLink menu*, page 804
- *Altia menu*, page 902

File menu

The **File** menu provides commands for creating or opening workspaces, loading projects, saving and printing, working with sequence files and analysis (static and dynamic), and exiting the Validator.

The menu also includes a numbered list of the most recently opened workspaces. To load one of them, choose it from the menu.



Menu commands

These commands are available on the menu:

New Workspace (Ctrl+N)

Creates a new workspace.

Open Workspace (Ctrl+O)

Displays a standard dialog box where you can open a workspace file.

Close Workspace

Closes the workspace. You will be asked whether to save any changes to files before they are closed.

Save Workspace

Saves the current workspace.

Save Workspace As

Displays a dialog box where you can save the current workspace with a new name.

Load Project

Displays a standard dialog box where you can open a new project.

Close Project

Closes the project.

Sequence File>New

Opens a new instance of the **Sequence File** window, see *Sequence File window*, page 383.

Sequence File>Open, Close, Save, Save As

Standard Windows command for opening, closing, and saving **Sequence File** windows.

Analysis>New Dynamic

Opens a new instance of the **Dynamic Analysis** window, see *Dynamic Analysis window*, page 375.

Analysis>New Static

Opens a new instance of the **Static Analysis** window, see *Static Analysis window*, page 388.

Analysis>Open, Close, Save, Save As

Standard Windows command for opening, closing, and saving **the Dynamic Analysis** and **Static Analysis** windows.

Print (Ctrl+P)

Prints the active document. Documents that can be printed are sequence files and static and dynamic analysis files.

***workspace*.vws (Ctrl+R)**

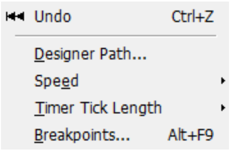
A numbered list of the most recently used workspaces, in reverse order of when they were last opened. Choose the one you want to open.

Exit (Alt+F4)

Exits the Validator. You will be asked whether to save any changes before the files are closed.

Edit menu

The **Edit** menu provides commands for editing.



Menu commands

These commands are available on the menu:

Undo (Ctrl+Z)

Undoes your most recent action.

Designer Path

Displays a dialog box where you can specify to the Validator where the Designer is installed. If you have not installed IAR Visual State in the default location, this information is required for the graphical animation.

Speed

Opens a submenu where you can set the time to pass between each step in the graphical animation.

Speed>Free Run

Plays the graphical animation as fast as possible.

Speed>Define

Displays the **Animation Speed** dialog box, see *Animation Speed dialog box*, page 365.

Timer Tick Length

Opens a submenu where you can set the tick length of timer ticks used in the Validator.

Timer Tick Length>Define

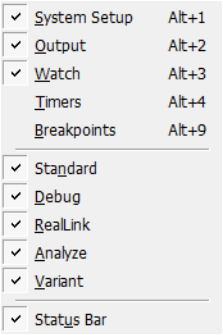
Displays the **Timer Tick Length** dialog box, see *Timer Tick Length dialog box*, page 392.

Breakpoints (Alt+F9)

Displays the **Breakpoint Setup** dialog box see *Breakpoints Setup dialog box : General*, page 371.

View menu

The **View** menu provides commands for opening windows, displaying toolbars, and zooming in windows.



Menu commands

These commands are available on the menu:

System Setup (Alt+1)

Opens the **System Setup** window, see *System Setup window*, page 391.

Output (Alt+2)

Opens the **Output** window, see *Output window*, page 382.

Watch (Alt+3)

Opens the **Watch** window, see *Watch window*, page 398.

Timers (Alt+4)

Opens the **Timers** window, see *Timers window*, page 392.

Breakpoints (Alt+9)

Opens the **Breakpoints** window see *Breakpoints window*, page 366.

Standard

Shows/hides the **Standard** toolbar.

Debug

Shows/hides the **Debug** toolbar.

RealLink

Shows/hides the **RealLink** toolbar.

Analyze

Shows/hides the **Analyze** toolbar.

Variant

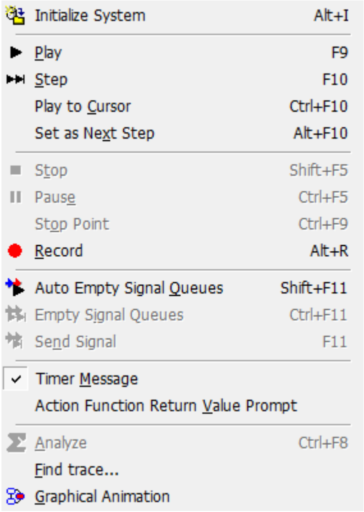
Shows/hides the **Variant** toolbar.

Status Bar

Shows/hides the status bar at the bottom of the Validator.

Debug menu

The **Debug** menu provides commands for simulating your state machine model.



Menu commands

Initialize System (Alt+I)

Initializes the system(s) to the startup state. This includes:

- Initializing the state configuration to State-Undefined
- Initializing all internal and external variables to their initial values
- Resetting the signal queue.

Play (F9)

Plays a recorded test sequence.

Step (F10)

Plays a recorded test sequence one step forward.

Play to Cursor (Ctrl+F10)

Plays a test sequence up to the selected command in a test sequence.

Set as Next Step (Alt+F10)

Sets the selected command in a test sequence as the next step to be processed.

Stop (Shift+F5)

Stops playing a recorded test sequence and resets it.

Pause (Ctrl+F5)

Pauses playing a recorded test sequence.

Stop Point (Ctrl+F9)

Stops playing a recorded test sequence at the selected command.

Record (Alt+R)

Starts recording a log sequence.

Auto Empty Signal Queues (Shift+F11)

Enables/disables automatic emptying of signal queues. When the command is enabled, the signal queue is automatically emptied when an event is sent manually to a system. During execution, the signal queue is not emptied automatically.

Empty Signal Queues (Ctrl+F11)

Sends all signals in all queues one signal at a time, beginning with the first queue, until all queues are empty.

Send Signal (F11)

Sends the first signal in the first queue that contains signals. Arrange the order of the queues in the **System Setup** window, see *System Setup window*, page 391.

Timer Message

Toggles whether or not a warning message is displayed when an event from a timer is about to be sent.

Action Function Return Value Prompt

Toggles whether or not you are prompted for action function return values.

Analyze (Ctrl+F8)

Starts an analysis, dynamic or static depending on the active analysis window.

Find Trace

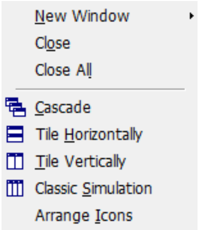
Displays the **Trace Setup** dialog box, see *Find Trace dialog box*, page 379.

Graphical Animation

Opens the Designer in Simulation Mode.

Window menu

The **Window** menu provides commands for arranging the Designer windows.



Menu commands

These commands are available on the menu:

New window>Systems (Ctrl+1)

Opens a new instance of the **Systems** window, see *Systems window*, page 389.

New window>Events (Ctrl+2)

Opens a new instance of the **Events** window, see *Events window*, page 377.

New window>Actions (Ctrl+3)

Opens a new instance of the **Actions** window, see *Actions window*, page 364.

New window>Variables (Ctrl+4)

Opens a new instance of the **Variables** window, see *Variables window*, page 395.

New window>Guard Expressions (Ctrl+5)

Opens a new instance of the **Guard Expressions** window, see *Guard Expressions window*, page 380.

New window>Signal Queues (Ctrl+6)

Opens a new instance of the **Signal Queues** window, see *Signal Queues window*, page 387.

Close

Closes the active window.

Close All

Closes all open windows.

Cascade

Arranges the open windows partially on top of each other but fanned out so that the window titles are visible.

Tile Horizontally

Changes the size of the open windows and arranges them from top to bottom so that they are all visible.

Tile Vertically

Changes the size of the open windows and arranges them from left to right so that they are all visible.

Classic Simulation

Arranges the windows of the Validator according to a default layout suitable for simulation.

Arrange Icons

Arranges minimized windows.

Help menu

The **Help** menu displays information about the Validator.

Validator shortcut key summary

General

These are the general shortcut keys:

Description	Shortcut key
Create a new workspace	Ctrl+N
Open a workspace	Ctrl+O
Save an open file	Ctrl+S
Stop a running timer	Delete
Open the online help system	F1
Exit the Validator	Alt+F4
Undo the latest action	Ctrl+Z

Table 20: General Validator shortcut keys

Windows

These are the shortcut keys for opening windows:

Description	Shortcut key
Open a new instance of the **Systems** window	Ctrl+1
Open a new instance of the **Events** window	Ctrl+2

Table 21: Validator windows shortcut keys

Description	Shortcut key
Open a new instance of the **Actions** window	Ctrl+3
Open a new instance of the **Variables** window	Ctrl+4
Open a new instance of the **Guard Expressions** window	Ctrl+5
Open a new instance of the **Signal Queues** window	Ctrl+6
Show the runtime model (only when in target mode)	Alt+F8
Open the **Field Chooser** window	Alt+0
Open the **System Setup** window	Alt+1
Open the **Output** window	Alt+2
Open the **Watch** window	Alt+3
Open the **Timers** window	Alt+4
Open the **Breakpoints** window	Alt+9

Table 21: Validator windows shortcut keys

Simulation

These are the shortcut keys for simulation:

Description	Shortcut key
Display the **Breakpoint Setup** dialog box	Alt+F9
Initialize all systems	Alt+I
Play a recorded test sequence	F9
Play a recorded test sequence one step forward	F10
Play a test sequence up to the selected command in a test sequence	Ctrl+F10
Set the selected command in a test sequence as the next step to be processed	Alt+F10
Stop playing a recorded test sequence and reset it	Shift+F5
Pause playing a recorded test sequence	Ctrl+F5
Stop playing a recorded test sequence at the selected command	Ctrl+F9
Starts recording a log sequence	Alt+R
Enable/disable automatic emptying of signal queues	Shift+F11
Send all signals in all queues one signal at a time, beginning with the first queue, until all queues are empty	Ctrl+F11
Sends the first signal in the first queue that contains signals	F11

Table 22: Validator simulation shortcut keys

Description	Shortcut key
Start an analysis, dynamic or static depending on the active analysis window	Ctrl+F8
Add an element to **Watch** window	Shift+F9
Go to the next test sequence	Ctrl+Down Arrow
Go to the previous test sequence	Ctrl+Up Arrow

Table 22: Validator simulation shortcut keys

Part 5. Formal verification using the Verificator

This part of the *IAR Visual State User Guide* includes these chapters:

- Formal verification
- Checks performed by the Verificator
- Verificator command line options

Formal verification

- Introduction to formal verification using the Verificator
- Verifying state machine models
- Graphical environment for the Verificator

Introduction to formal verification using the Verificator

Learn more about:

- *Briefly about verification using the Verificator*, page 413
- *The checks that can be performed—an overview*, page 413
- *Verification modes*, page 416
- *Verification strategies*, page 416
- *Optimizing for verification*, page 420

BRIEFLY ABOUT VERIFICATION USING THE VERIFICATOR

The Verificator uses formal verification to analyze Visual State systems. The Verificator creates a formal description of a system and establishes its properties using formal semantics. Verification results generated by the Verificator are, thus, 100% certain, just like mathematical theorems. The formal semantics of a system is described in terms of its runtime configurations, the so-called *state space*.

Verification with the Verificator is characterized by this:

- Formal verification: the logical consistency of a Visual State project is checked. The Verificator does not test functionality, in contrast to the Validator.
- Checks of complex properties such as state dead ends.
- Complete examinations of models with large state spaces.
- Computing traces that show how a model might reach a state in which a warning or error condition holds true.

THE CHECKS THAT CAN BE PERFORMED—AN OVERVIEW

The Verificator analyzes of the behavior of your system to check its logical consistency. During the analysis, your state machine model is placed in an environment where any sequence of events is possible. If the model is consistent in this most extreme environment, the model is consistent in all possible real-world environments. Checking for logical consistency means checking these aspects:

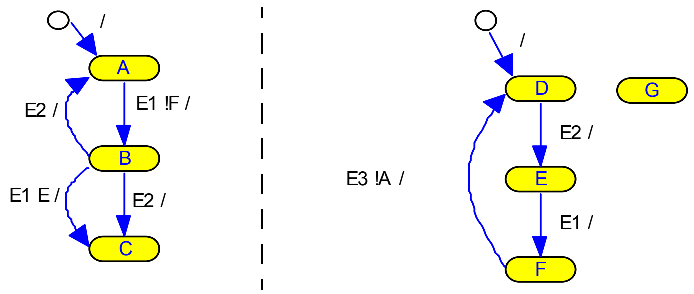
- Are all elements used?
- Are all elements activated?
- Are there any ambiguities, such as conflicting transitions or dynamic ambiguous assignments?
- Does the system contain any dead ends?
- Is the signal queue neither too short nor too long?
- Is there any possibility of underflow, overflow, or similar arithmetic errors occurring?

If a critical error is detected during verification, your system contains logical errors. You are recommended not to code-generate a system that contains critical errors.

For information about all available checks, see *Checks performed by the Verificator*, page 433.

An example verification

Consider the state machines in this system:



If you run the Verificator 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 a 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 in this example 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 in this example 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 states. The system in this example has the following local dead ends:

  ```
  Local dead end for the machine: R0
  {topState.A, topState.C} x {topState.F}
  {topState.C} x {topState.D, topState.E}

  Local dead end for the machine: R1
  {topState.A} x {topState.F}
  ```

- System dead ends

  System dead ends are state configurations that prevent all the state machines in the system from changing states. The system in this example 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. This transition in the example is never triggered:

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

Thus, the transition can be removed without changing the behavior of the model.

Never activated elements and dead ends might or might not indicate errors in the system, when they are reported as warnings. In contrast to that, conflicting transitions are always an error and are reported as such.

For a list of the warnings and error messages given by the Verificator, see *Overview of checks, modes, and errors*, page 433.

VERIFICATION MODES

The Verificator can run in two modes, *Full Forward* (used by default) and *Full Compositional mode*.

These two modes apply to exactly the same models and they check the exactly same properties (except that the compositional mode cannot detect state and system dead ends).

However, the two modes differ in their performance characteristics. The Forward mode is faster than the Compositional mode on some models, but slower on other models. You will have to find out by experimenting which mode is the faster one on a specific model.

The Full Forward mode

The Forward mode takes a global view of the model, starting out from the global initial state and then iterating over the entire state space.

The Full Compositional mode

The Compositional mode performs not one but many state space iterations, one for every property to be checked. These iterations proceed backwards from the states that satisfy this property and ignore the parts of the model that are irrelevant for the property. As a consequence, the Compositional mode tends to work best on models that are "loosely coupled"—with few signals and other dependencies between the model components.

Note: Because this mode iterates over the state space in backward direction, compositional mode is sometimes called compositional backwards mode.

VERIFICATION STRATEGIES

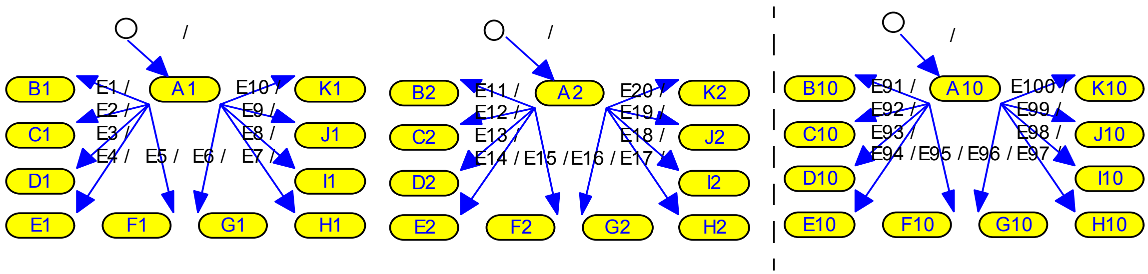
The Verificator uses the following strategies.

Formal verification on large systems

The verification results are found by the Verificator after examining the complete state space of a Visual State System.

The Verificator represents 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:



The system in this example consists of ten state machines of ten states each, which means that there are about 10^10 configurations of the entire system. Any of the state configurations can be reached in no more than about 10 steps. Symbolic state space exploration can explore the entire state space in the same number of steps. In contrast, using a simulation tool for checking this system is clearly not possible because the state space is too large—stepping through each configuration individually would require extremely long time.

The Verificator starts out at the initial state configuration, proceeding step-by-step to explore all possible forward transitions. Thus, symbolic state space exploration can cover the entire state space in a number of steps equal to the maximum step distance of any state configuration to the initial configuration.

Non-verifiable elements

State machine models are not verifiable if they contain elements of the type `VS_FLOAT` or `VS_DOUBLE`. In the Designer you can set a safe mode option to be given a warning when you create or use non-verifiable elements during model design, see *Getting warnings for non-verifiable elements*, page 233.

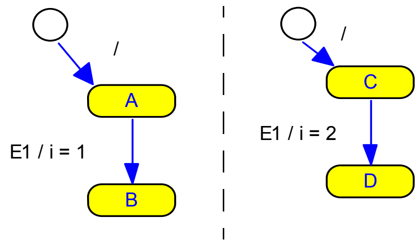
Systems with ambiguous behavior

The UML standard 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 Visual State systems are ambiguous, and in such cases the Verificator will give an error message.

Note: Assignments that belong to the same execution step never lead to any ambiguity as long as every variable written to is written to only once.

Example 1

The system in this example consists of two state machines. The assignments to `i` are ambiguous, which will be detected by the Verificator:

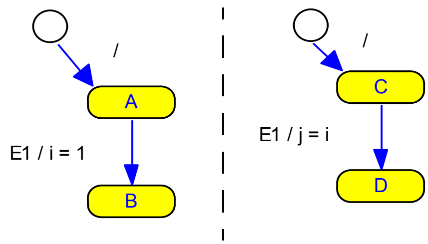


The system is ambiguous because the sequence in which the transitions will be triggered is not specified. Which system configuration should be entered after the event `E1?: (B, D, i = 1)`, `(B, D, i = 2)`, or maybe `(B, D, i = 3)`?

Ambiguity makes the model ill-defined, which means that the Verificator must be re-run once the ambiguity has been solved.

Example 2

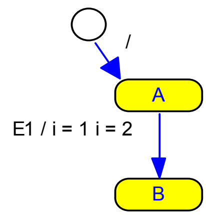
In this example, the assignment to `j` is ambiguous because it might read either the old or the new value of `i`, which is assigned to in the same microstep:



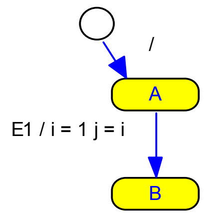
Example 3

In this example, the state machines in system `a` and `b` have ambiguous behavior:

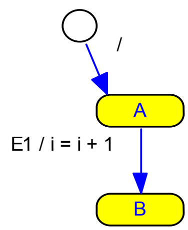
a:



b:



c:



Multiple assignments to the same variable result in an ambiguity. An example of that is system `a` because `i` is assigned to twice on the same transition. The systems `b` and `c` are unambiguous because the buffered value of `i` is read in the assignments `j = i` or `i = i + 1`, respectively.

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++ as long as no wrap-around occurs. Wrap-arounds as treated by the Verificator might be different from wrap-arounds on target because C/C++ evaluates expressions using `int` or `long int` arithmetic, whereas the Verificator evaluates expressions using the stated domains of their operands. The Verificator reports wrap-arounds as under- or overflow warnings. If wrap-arounds might lead to discrepancies between verification results and model behavior on target, you should make sure to modify the model to do away with under- or overflow warnings. 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.

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.

See also *Non-verifiable elements*, page 417.

OPTIMIZING FOR VERIFICATION

Verification can be very time-consuming and require extensive memory resources. Here follow some guidelines for efficient use of time/memory managing options and some recommendations on modeling. The constructs that should preferably be avoided to verify your system are also listed.

Using time/memory options to help verification

Different combinations of verification options might be used if verification takes too much time or requires too much memory. However, note that verification times of over one hour are not unusual. Here are some guidelines:

- Verification mode—Full Forward or Full Compositional. Full Compositional verification can often handle very large systems, but is most suitable for systems that consist of many independent state machines (in other words, state machines that do not use signals and only use state conditions sparingly). See also *Verification modes*, page 416. Experiment to determine which mode that suits your design model best.
- Alternative verification heuristics—use this option to change the used heuristics, which might affect the verification time. Experiment to determine which setting that works best for your state machine model.
- Control variable ranges in assignments—use this option to make the Verificator try to exploit out-of-range conditions in variable assignments, which might affect the verification time. Experiment to determine which setting that works best for your state machine model. If you use this option, make sure to match Visual State data types and value ranges as closely as possible to the actual runtime values.
- Node space size—use this option to change the size of the node space, which is the memory area used for the data structures built during a verification. It is impossible beforehand to find the necessary size of the node space, so the right size must be found by experimenting. 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 a way that can lead to unnecessary memory swapping. Normally, the

Verificator can handle the node space requirement itself, but for large systems it can be beneficial to set the initial size of the node space manually. The option in the Navigator is **Size of node space**. The size of the node space is measured in bytes. Each node occupies 20 bytes.

- Skipping parts of verification—whenever a complete run is impossible, you can skip parts of the verification and verify only as much as possible:
  - In Full Compositional mode you can skip a verification check by specifying a timeout that applies to all checks, or by clicking **Skip** during verification.
  - In Full Forward mode you can stop state space exploration by clicking **Skip** during verification. Model properties are then computed based on the partially computed state space.

Note: Not all combinations of options are possible, because the setting for one option might limit the choices for other options. Hover over the option in the options dialog box with the mouse pointer to get information about any limitations.

Keeping down the complexity of verifying systems

It is possible to design Visual State 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 guidelines to keep down the complexity of verifying your systems, and thereby reduce time consumption:

- Signals and signal queues

  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, but it should not overflow. See *Check for signal queue size*, page 444.
- Operators
  - Do not use these operators with variables larger than 8 bits: `*`, `/`, `%`, `<<`, `>>`.
  - The bit size of variables that are 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.
  - 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 Verificator 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.

See also *Variables, domains, and arithmetics*, page 419.

Verifying state machine models

What do you want to do:

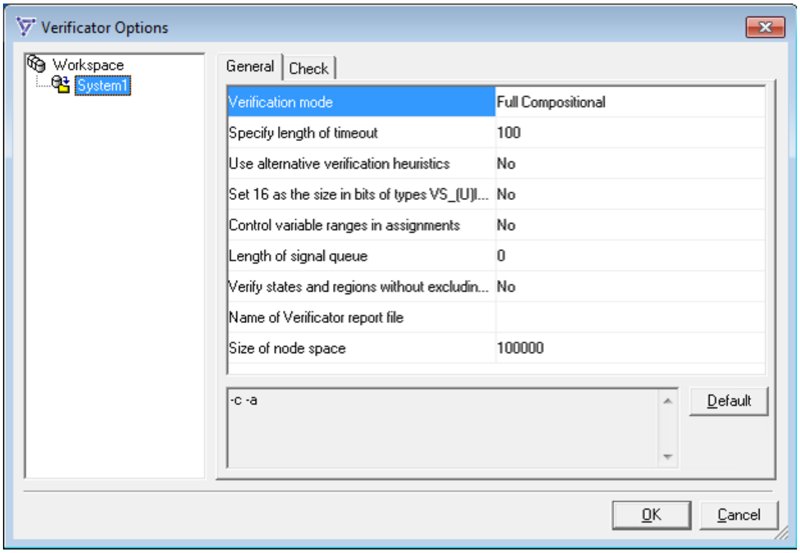
- *Starting the verification*, page 422
- *Tracing your verified state machine model*, page 425

For information about starting the Verificator from the command line, see *Invocation syntax for the Verificator*, page 447.

STARTING THE VERIFICATION

1 In the Navigator, open your workspace file.

2 Choose **Project>Options>Verification** to open the **Verificator Options** dialog box.



For reference information, see *Verificator Options dialog box*, page 426. For a general description of how to set options, see *Setting Verificator, Coder, and Documenter options*, page 79.

3 In the tree browser to the left, select the system for which to set options.

4 On the **General** page, set general Verificator options. On the **Check** page, select the checks to be performed.

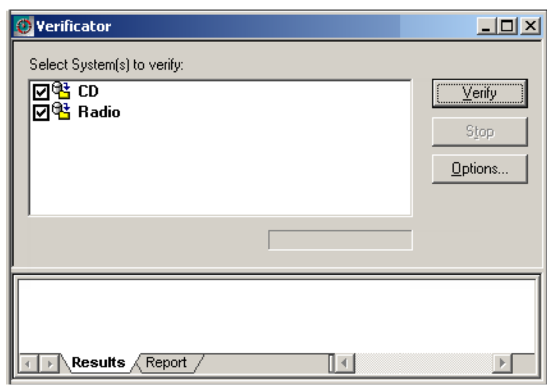
Some option combinations might lead to very extensive verification, which might take extremely long time to perform. If that happens, try a different combination of options. For guidelines, see *Optimizing for verification*, page 420.

Note: Not all combinations of options are possible, because the setting for one option might limit the choices for other options. Hover over the option with the mouse pointer to get information about any limitations.

Click **OK** when your are finished setting options.

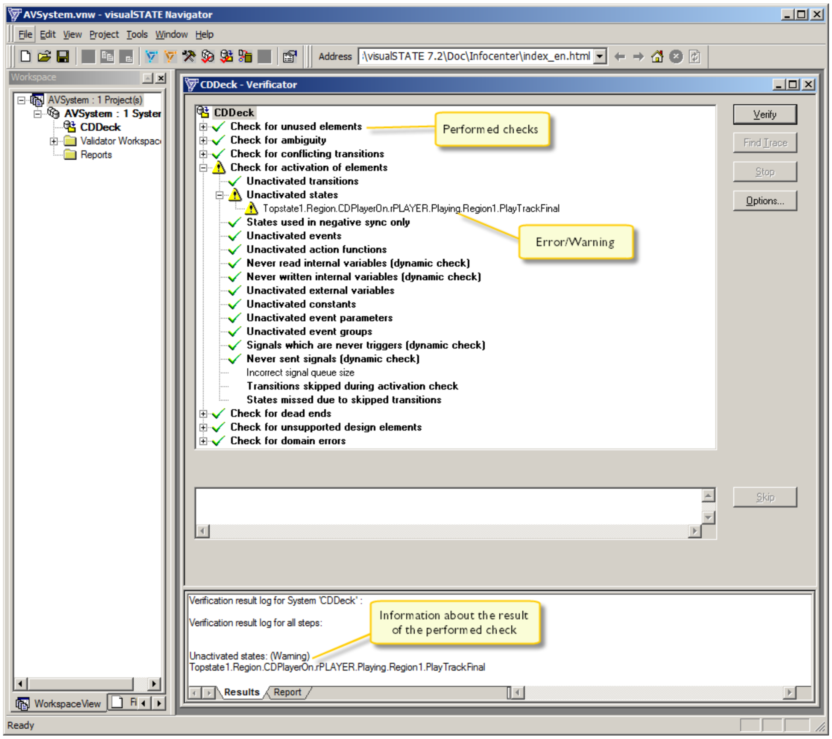
5 On the **Project** menu, choose **Verify System** or **Verify Multiple Systems**, whichever is relevant for you.

If there is more than one system in the project, and you choose **Verify Multiple Systems**, a dialog box is displayed where you can select the system(s) to verify.



Select the appropriate system(s) and click **Verify**.

6 A verification progress window is displayed. Information is listed by groups of checks:



The window provides an immediate view of the results of the verification. Performed checks are highlighted in bold (in the upper part of the window). Checks that have resulted in errors or warnings are marked.

To see the cause of a warning or an error, select the check; information is displayed on the **Results** page.

To view the result for an entire system, select the system in the upper part of the window.

7 To change the Verificator options, select the system in the tree browser in the **Verificator** window, and click the **Options** button. Make your changes and click the **Verify** button.

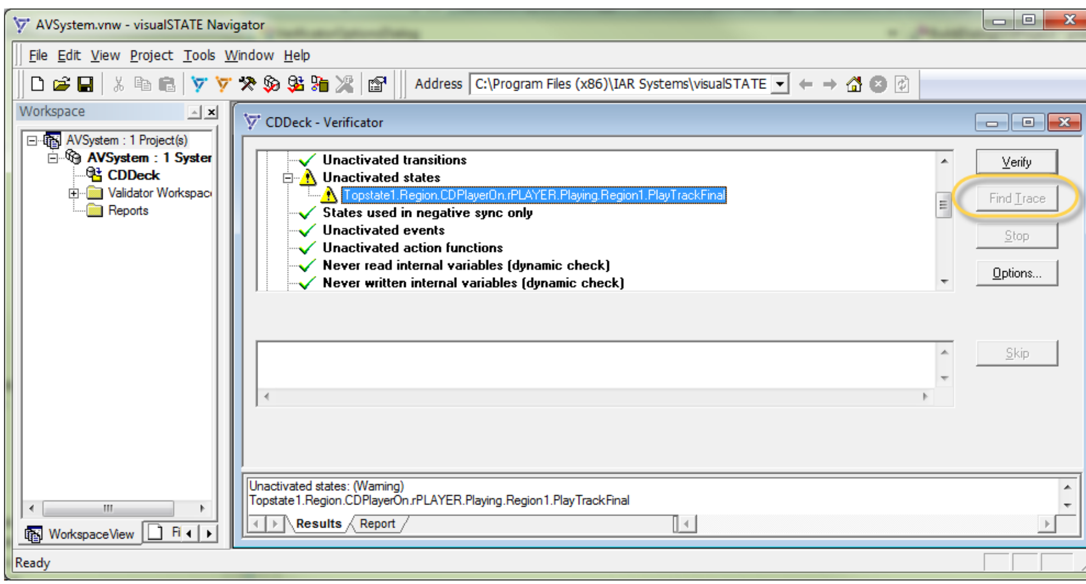
If you selected **Yes** for **Write Verification report** in the **Options** dialog box, you can view a summary of the completed verification on the **Report** page.

TRACING YOUR VERIFIED STATE MACHINE MODEL

A trace is a sequence of steps that will take the system to a specific state configuration. The trace is saved in a sequence file.

1 Before you can perform a trace in the Navigator, your must first run a verification, see *Starting the verification*, page 422.

2 In the **Verificator** window, select the state flagged with an error or warning that you want to trace to. If the error or warning can be traced, the **Find Trace** button is enabled.



3 Click the **Find Trace** button.

4 In the dialog box that is displayed, specify the name and the location for the trace output file. Click **Save**.

5 The Verificator performs the trace to the error or warning you specified. This trace is stored in a test sequence file. After the test sequence file has been saved, the Validator will be opened with the file loaded. See *Recording and playing test/event sequences*, page 349.

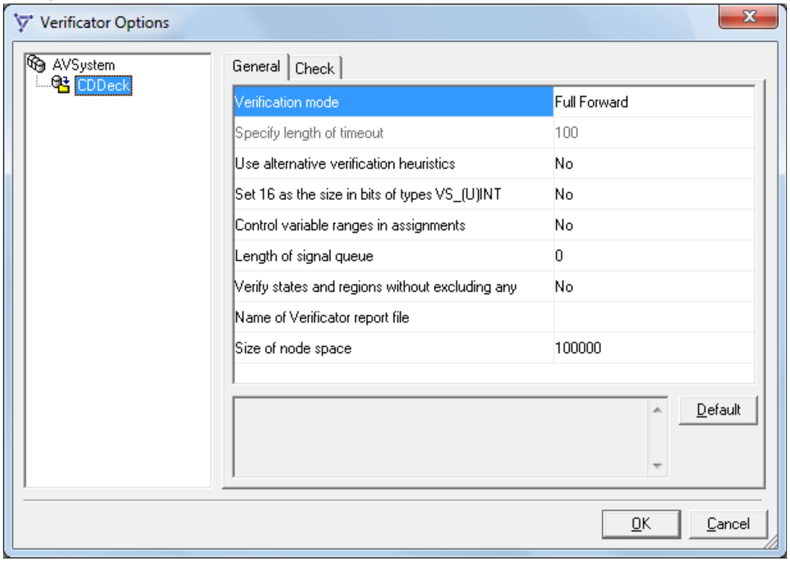
Graphical environment for the Verificator

Reference information about:

- *Verificator Options dialog box*, page 426
- *Verificator window*, page 430

Verificator Options dialog box

The **Verificator Options** dialog box is available from the **Project** menu in the Navigator.



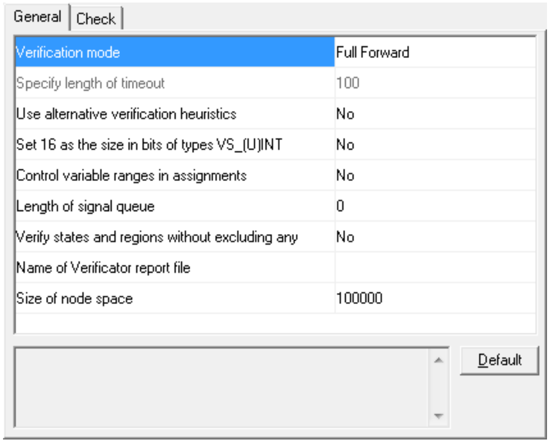
Use this dialog box to set options for the Verificator.

You can set options on these tabbed pages:

- *Verificator Options : General*, page 427
- *Verificator Options : Check options*, page 429

Verificator Options : General

The **General** options page contains general options.



Use this page to make general settings for the Verificator. The display area under the options shows the resulting command line for the verification.

Verification mode

The Verificator can run in two modes. The two modes differ in their performance characteristics. You must experiment to find out which mode is the faster one on a specific model. For more information, see *Verification modes*, page 416.

Choose between:

Full Forward

The Full Forward verification mode will be used.

Full Compositional

The Full Compositional verification mode will be used. In this mode, the Verificator cannot detect state or system dead ends.

Specify length of timeout

Specify the length of the timeouts used in the Full Compositional mode. When a timeout occurs, the verification will skip the goal and continue with the next goal.

Use alternative verification heuristics

Determines whether alternative heuristics are used for the verification. Experiment to determine which setting works best for a given model.

Set 16 as the size in bits of types VS_(U)INT

Controls the size of the `VS_INT` and `VS_UINT` types. Choose between:

Yes

The size of the `VS_INT` and `VS_UINT` types is 16 bits.

No

The size of the `VS_INT` and `VS_UINT` types is 32 bits

Control variable ranges in assignments

Determines whether the Verificator will try to exploit out-of-range conditions in variables.

Length of signal queue

Specify the length of the signal queue to use. The length influences the complexity of verification.

Verify states and regions without excluding any

Determines whether states and regions marked for exclusion in the Designer are included or excluded from verification. Choose between:

Yes

Verifies all states and regions, including those marked for exclusion in the Designer.

No

Excludes states and regions marked for exclusion in the Designer from verification.

Write Verificator report

Determines whether the verification report is written to a text file or not.

Name of Verificator report file

Specify the name of the verification report file if the report is saved to a text file.

Size of node space

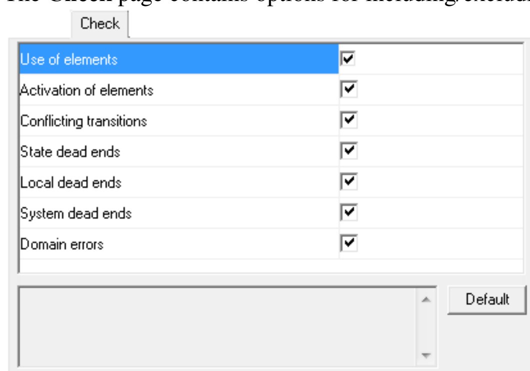
Specify the default initial size of node space in bytes. Larger node space usually yields quicker verification.

Default

Restores the options to their default settings.

Verificator Options : Check options

The **Check** page contains options for including/excluding certain checks.



Use this page to control which checks that the Verificator should perform. The display area beneath the options shows the resulting command line for the verification.

Use of elements

Includes/excludes the use of elements from being checked by the verification.

Activation of elements

Includes/excludes the activations of elements from being checked by the verification.

Conflicting transitions

Includes/excludes transition conflicts from being checked by the verification.

State dead ends

Includes/excludes state dead ends from being checked by the verification. Note that the Verificator cannot detect state dead ends in Full Compositional mode.

Local dead ends

Includes/excludes local dead ends from being checked by the verification.

System dead ends

Includes/excludes system dead ends from being checked by the verification. Note that the Verificator cannot detect system dead ends in Full Compositional mode.

Domain errors

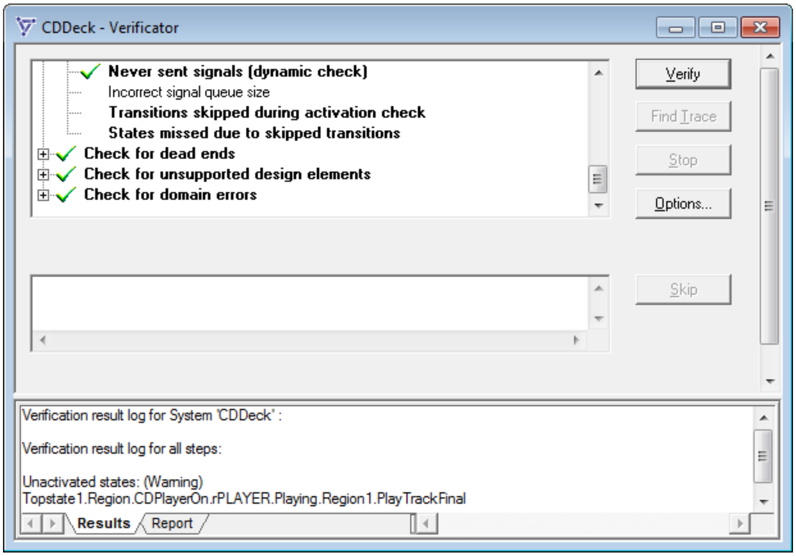
Includes/excludes domain errors from being checked by the verification.

Default

Restores the options to their default settings.

Verificator window

The **Verificator** window is available from the **Project** menu in the Navigator.



This window contains the graphical interface to the Verificator.

Log area

Displays the results of the verification. Items selected for verification are shown in bold. To see the cause of a warning or an error, select the item and read the error or warning text in the **Output** pane. If an error or warning can be traced, the **Find Trace** button is enabled.

Verify

Performs a new verification.

Find Trace

Displays a standard Windows dialog box where you navigate to or create a trace output file. When you have done that, the Navigator starts the Verificator to try to find a trace to the error or warning and saves the result. The output file will be opened in the Validator, see *Recording and playing your test sequences*, page 350.

Stop

Stops the verification.

Options

Displays the **Verificator Options** dialog box, see *Verificator Options dialog box*, page 426.

Progress display

Displays the progress of the ongoing verification.

Skip

Skips the verification step that is currently being executed. Whenever a complete run is impossible, you can skip parts of the verification and verify as much as possible.

Output pane

This pane has two tabbed pages:

Results

Displays a description of the selected verification step in the **Log area**.

Report

Displays the detailed verification report, if you have set the **Write Verificator report** option in the **Verificator Options** dialog box to **Yes**.

Checks performed by the Verificator

- Overview of checks, modes, and errors
- Performing various checks

Overview of checks, modes, and errors

This table lists the Verificator checks performed in the two modes, and whether the errors given in the check report should be considered critical errors:

Check for	In Full mode	In Compositional mode	Considered critical
Unused elements			
States	Yes	Yes	No
Variables, event parameters, constants, enumerators	Yes	Yes	No
Action functions	Yes	Yes	No
Events, event groups, and signals	Yes	Yes	No
Transitions	Yes	Yes	No
Conflicting transitions	Yes	Yes	Yes
State dead ends	Yes	No	No
Local dead ends	Yes	Yes	No
System dead ends	Yes	No	No
Dynamic ambiguous assignments	Yes	No	Yes
Static ambiguous assignments	Yes	Yes	Yes
Signal queue size	Yes	Yes	Yes †
Domain errors	Yes	Yes	No

Table 23: Verificator checks, modes, and errors

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

Performing various checks

These checks are available:

Check for unused elements

Why perform this check

To identify elements that will never be used.

Description

The Verificator performs a static analysis of a system to check whether all declared elements are used. These elements are checked:

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

States

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

Variables, event parameters, constants, and enumerators

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, constants, and enumerators 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.

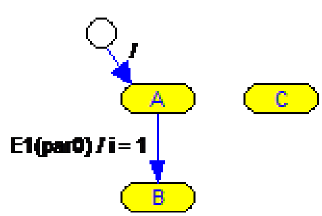
Events, event groups, and signals

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



This state machine in this system has these elements defined:

Events:	`E1(VS_INT ar0), E2`
Internal variable:	`i`
External variable:	`x`
Signal:	`S1`

Performing a Verificator check on the system gives the following result for unused elements:

```
Never read internal variables (static check): (Warning)
i

Unused external variables: (Warning)
x

Unused event parameters: (Warning)
E1.par0

Unused events: (Warning)
E2

Unused states: (Warning)
Topstate1.Region1.C

Signals which are never triggers (static check): (Warning)
S1

Unactivated states: (Warning)
Topstate1.Region1.C

Unactivated events: (Warning)
E2

Never read internal variables (dynamic check): (Warning)
i

Unactivated external variables: (Warning)
x

Unactivated event parameters: (Warning)
par0

Signals which are never triggers (dynamic check): (Warning)
S1

Never sent signals (dynamic check): (Warning)
S1
```

Check for activation of elements

Why perform this check

To identify elements that will never be activated.

Description

The check for activation of elements is similar to the check for unused elements, 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 system. A transition is said to be *reachable* if a sequence of events can lead to the transition being triggered.

These elements are checked for activation:

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

States

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

Variables, event parameters, constants, and enumerators

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 a 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 a parameter for any activated action function.

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

Event parameters, constants, and enumerators are reported as never activated if they are not read in any activated guard expression or assignment, or used as a 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.

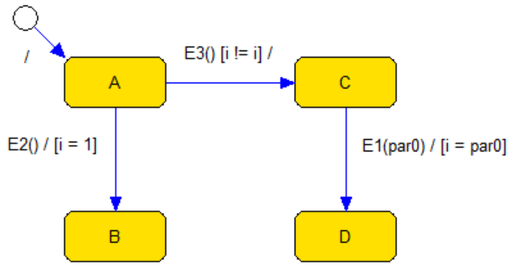
Signals that are not used on the transition action side of any reachable transition are reported as never sent.

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

Transitions

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

Example



This system has these elements defined:

Events:	`E1(VS_INT par0), E2, E3`
Internal variable:	`i`
External variable:	`x`
Signal:	`S1`

Performing a Verificator check on the system gives the following result for never activated elements:

```
Unused external variables: (Warning)
x
```

```
Signals which are never triggers (static check): (Warning)
S1

Never sent signals (static check): (Warning)
S1

Unactivated transitions: (Warning)
A: E3() [i != i] / -> C
C: E1(par0) / [i = par0] -> D

Unactivated states: (Warning)
Topstate1.Region1.C
Topstate1.Region1.D

Unactivated events: (Warning)
E1
E3

Unactivated external variables: (Warning)
x

Unactivated event parameters: (Warning)
par0

Signals which are never triggers (dynamic check): (Warning)
S1

Never sent signals (dynamic check): (Warning)
S1
```

Check for conflicting transitions

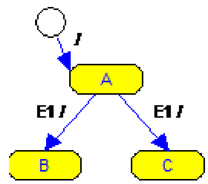
Why perform this check

To identify conflicting transitions.

Description

Two transitions with a 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



Performing a Verificator check on the system reports the following result for conflicting transitions:

```
The following transitions conflict:
A:
  E1() /
-> B
A:
  E1()
-> C
```

Check for state dead ends

Why perform this check

To identify state dead ends.

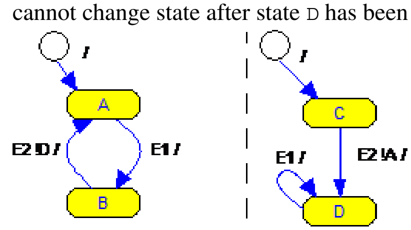
Description

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

Note: This check is only performed in Full Forward mode.

Example

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 reports the following result for state dead ends:

```
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.

Check for local dead ends

Why perform this check

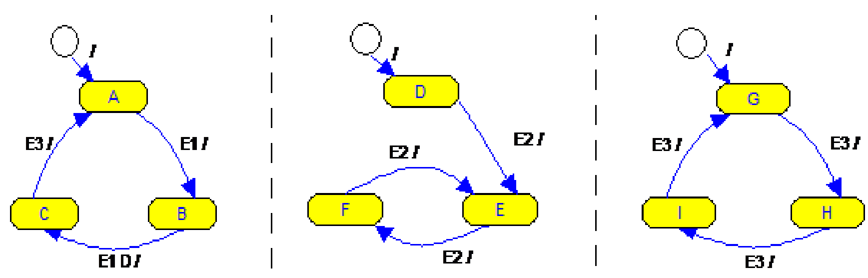
To identify local dead ends.

Description

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

Example

The system contains three state machines, `R0`, `R1`, and `R2` from left to right:



The first machine deadlocks when the system enters the state configurations `(B, F)` and `(B, E)`.

Performing a Verificator check on the system reports the following result for local dead ends:

```
Local dead ends for the machine: R0
{B} x {E, F} x {*}
```

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

Check for system dead ends

Why perform this check

To identify system dead ends.

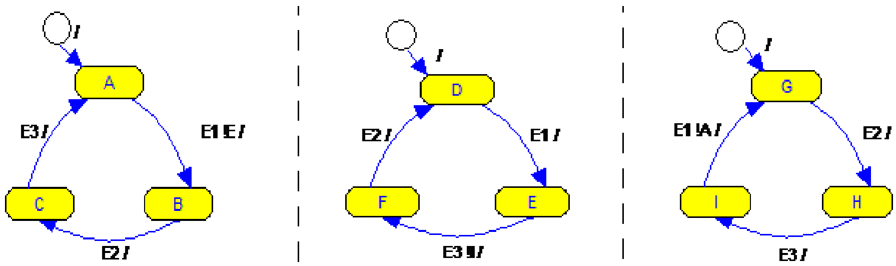
Description

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

Note: This check is only performed in Full Forward mode.

Example

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 reports the following result for system dead ends:

```
System dead ends
{A} x {E} x {I}
```

The system dead end can be reached by the event sequence `E2, E3, E1, E2, E3`.

Check for dynamic ambiguous assignments

Why perform this check

To identify dynamic ambiguous assignments.

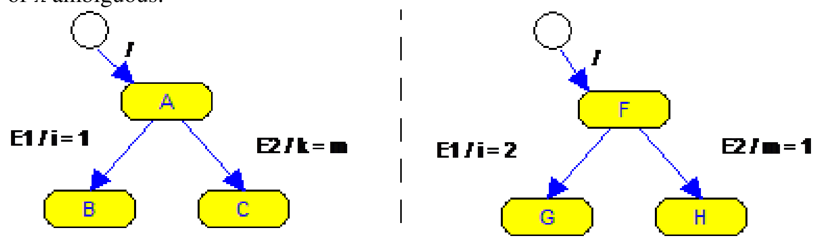
Description

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.

Note: This check is only performed in Full Forward mode.

Example

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 Verificator check on the system reports the following ambiguity result:

```
Ambiguous assignments (dynamic check): (Error)
The variable i is assigned several times on the transitions
A: E1() / [i = 1] -> B

  and

D: E1() / [i = 2] -> E
```

Check for static ambiguous assignments

Why perform this check

To identify static ambiguous assignments.

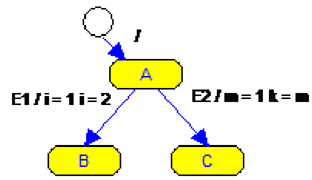
Description

When there are multiple assignments on a single transition, they are executed in some fixed sequence in the code generated by the Visual State Coder. However, such assignments cannot be handled in a Full Forward 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 Forward mode.

Note: This check is only performed in full forward mode.

Example

The transition `A -> B` assigns `i` twice. The transition `A -> C` both reads and writes `m`.



Performing a Verificator check on the system reports the following static ambiguity results:

```
The variable i is assigned several times on the transition
A:
  E1() / [i = 1] [i = 2]
—> B
```

Check for signal queue size

Why perform this check

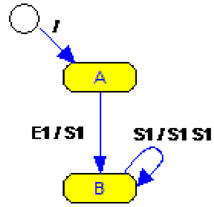
To identify the optimal signal queue size.

Description

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 signal queue size. 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 the signal queue behavior and size*, page 190. Signal queue overflow is an error which means that the remaining part of the verification will be based on false assumptions.

Note: Systems that need an unbounded signal queue cannot be fully verified, which the following example system illustrates.

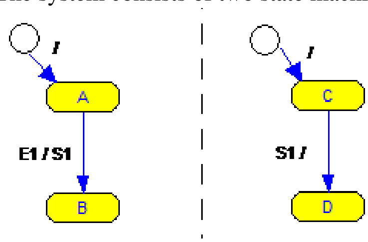
The system will continue to add signals to the signal queue until the queue overflows, resulting in incorrect verification.



See also *Signal*, page 181 and *Signal queue*, page 181.

Example

The system consists of two state machines.



Performing a Verificator check on the system reports the following information about the signal queue size:

- 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`

Check for domain errors

Why perform this check

To identify arithmetic error conditions.

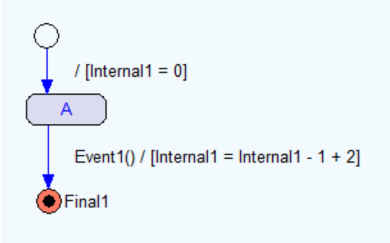
Description

Your system should not exhibit any arithmetic error conditions. The Verificator can check for these problems:

- Range errors—assigning an out-of-range value to variables
- Arithmetic over- or underflows—performing a calculation whose result lies outside the range of admissible values
- Array subscription errors
- Divisions by zero
- Shifting errors—shifting by more than the operand's length.

Example

This system exhibits an arithmetic underflow, given that the variable `Internal1` has been declared as being of type `UInt32`.



Performing a Verificator check on the system reports these domain errors:

```
Range error, overflow, underflow, subscription error, division by
zero, or shifting error: (Warning)
M_Topstate1_Region1:
In component: Internal1 = Internal1 - 1 + 2
Of rule: State1: Event1() / [Internal1 = Internal1 - 1 + 2] ->
Final1
When evaluated in control flow state(s):
{Topstate1.Region1.State1}
```

Verificator command line options

- Introduction to invoking the Verificator using command line options
- Summary of Verificator options
- Descriptions of Verificator options

Introduction to invoking the Verificator using command line options

Learn more about:

- *Briefly about invoking the Verificator*, page 447
- *Invocation syntax for the Verificator*, page 447

BRIEFLY ABOUT INVOKING THE VERIFICATOR

You can set Verificator options either in the Navigator—using the **Verificator Options** dialog box—or via the command line. For each option available in the **Verificator Options** dialog box, there is an equivalent option for the command line.

INVOCATION SYNTAX FOR THE VERIFICATOR

This is the invocation syntax for starting the Verificator from the command line:

```
Verificator.exe project_file system_name [option]...
```

Example 1

```
Verificator.exe Example.vsp VS_System -v
```

Description:	Verifies the system `VS_System` in the project file `Example.vsp` and writes the result to the screen.

Example 2

```
Verificator.exe Example.vsp VS_System -xlocal_dead_ends
-vReport.txt -c -s4
```

Description: Verifies the system `VS_System` in the project file `Example.vsp` in Compositional mode using a signal queue of length 4. Excludes checking for local dead ends. Writes the result to the file `Report.txt`.

Example 3

```
Verificator.exe Example.vsp VS_System -tOut.vlg
-dsTopstate.StateA
```

Description: Performs 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`.

Summary of Verificator options

This table summarizes the Verificator command line options:

Command line option	Description
`-B`	Makes all variables be treated as signed integers.
`-c`	Verifies in Compositional mode.
`-ds`	Specifies the destination state of a trace.
`-f`	Verifies all states and regions, including regions and states marked as excluded.
`-large|-Large`	Minimizes the memory consumption at the expense of a longer verification time.
`-p`	Uses the Verificator options specified in the Navigator.
`-s`	Overrides the length of the signal queue.
`-S`	Specifies the initial size of the node space.
`-small|-Small`	Minimizes the verification time at the expense of a larger memory consumption.
`-t`	Specifies which file that the trace should be saved in.
`-u`	Controls variable ranges in assignments to exploit out-of-range conditions.

Table 24: Verificator command line options

Command line option	Description
`-v`	Writes the verification report to a text file.
`-variant`	Specifies which variant to verify.
`-w`	Specifies the size of the `VS_INT` and `VS_UINT` types as 16 bits.
`-x`	Excludes a check from the verification.
`-y`	Makes the Verificator use alternative verification heuristics.

Table 24: Verificator command line options

Descriptions of Verificator options

The following pages give detailed reference information about each Verificator command line option.

-B

Syntax
`-B`*`size`*

Parameters

`size`	The size in bits.

Description
Makes all variables be treated as signed integers encoded in *`size`* bits.

This option is not available in the graphical interface.

-c

Syntax
`-c`

Description
Verifies in Compositional mode.

See also
Verification modes, page 416.

This option is used by default in the graphical interface.

-ds

Syntax `-dsstate`

Parameters

state The name of the destination state of the trace.

Description Specifies the destination state of a trace. This option can be repeated to add more states.

See also *Tracing*, page 341.

This option is not available in the graphical interface.

-f

Syntax `-f`

Description Verifies all states and regions, including regions and states marked as excluded.

Project>Options>Verification>General>Verify states and regions without excluding any

-large

Syntax `-large|-Large`

Description Minimizes the memory consumption at the expense of a longer verification time. This option is suitable for large systems.

See also *Using time/memory options to help verification*, page 420.

This option is not available in the graphical interface.

-p

Syntax `-p`

Description Uses the Verificator options specified in the Navigator.

See also *Verificator Options dialog box*, page 426.

This option is used by default in the graphical interface.

-s

Syntax `-ssize`

Parameters

size	The size of the signal queue.

Description Verifies using a signal queue of size *size*. If this option is not used, the size of the signal queue is set to the value specified in the project file.

See also *Keeping down the complexity of verifying systems*, page 421.

Project>Options>Verification>General>Length of signal queue

-S

Syntax `-Ssize`

Parameters

size	The initial size of the node space.

Description Specifies the initial size of the node space. A larger node space usually leads to quicker verification.

See also *Using time/memory options to help verification*, page 420.

Project>Options>Verification>General>Size of node space

-small

Syntax `-small|-Small`

Description Minimizes the verification time at the expense of a larger memory consumption. This option is suitable for small systems.

See also *Using time/memory options to help verification*, page 420.

This option is not available in the graphical interface.

-t

Syntax `-tfile`

Parameters

file	The name and path of the output file.

Description Specifies which file and path that the trace should be saved in (typically with a `vxlg` filename extension).

See also *Tracing*, page 341.

This option is not available in the graphical interface.

-u

Syntax `-u`

Description Controls variable ranges in assignments to exploit out-of-range conditions, if possible, and speed up the verification. If a range error is detected in an assignment, a fixed constant value is assigned to the variable on the left-hand side.

See also *Using time/memory options to help verification*, page 420.

Project>Options>Verification>General>Control variable ranges in assignments

-v

Syntax `-vfile`

Parameters

file	The name of the output file.

Description Writes the verification report to a text file.

Project>Options>Verification>General>Name of Verificator report file

-variant

Syntax `-variant`*`name`*

Parameters

`name`	The name of the variant.

Description Specifies which variant to verify. By default, the Verificator verifies the complete model.

See also *Using variants and features*, page 217.

Use the **Variant** toolbar.

-w

Syntax `-w`

Description Specifies the size of the `VS_INT` and `VS_UINT` types as 16 bits. By default, these types are 32 bits.

Project>Options>Verification>General>Set 16 as the size in bits of types VS_(U)INT

-x

Syntax `-x`*`check`*

Parameters The parameter *`check`* can be one of the following:

`activation`	Excludes activations of elements from being checked.
`conflicts`	Excludes conflicting transitions from being checked.
`domain_errors`	Excludes domain errors from being checked.
`local_dead_ends`	Excludes local dead ends from being checked.

`state_dead_ends`	Excludes state dead ends from being checked. This parameter has no effect in Full Compositional mode.
`system_dead_ends`	Excludes system dead ends from being checked. This parameter has no effect in Full Compositional mode.
`use`	Excludes the use of elements from being checked.

Description

Excludes the specified check from the verification. This option can be repeated to exclude more checks.

To set these options, choose:

Project>Options>Verification>Check

-y

Syntax

`-y`

Description

Makes the Verificator use alternative verification heuristics. This makes verification faster for some models.

See also

Using time/memory options to help verification, page 420.

Project>Options>Verification>General>Use alternative verification heuristics

Part 6. Code generation using a Coder

This part of the *IAR Visual State User Guide* includes these chapters:

- Code generation
- HCoder API code generation
- HCoder API reference information
- The Visual State Hierarchical Coder
- Hierarchical Coder command line options
- Adaptive API code generation
- Uniform API code generation
- Adaptive API reference information
- Uniform API reference information
- The Visual State Classic Coder
- Classic Coder command line options

Code generation

- Introduction to code generation, the Coders, and the APIs
- Generating code using a Coder and an API

Introduction to code generation, the Coders, and the APIs

Learn more about:

THE HIERARCHICAL CODER VERSUS THE CLASSIC CODER

IAR Visual State offers a choice of two code generators, the Visual State Classic Coder (called just "the Coder" in previous versions of IAR Visual State) and the Visual State Hierarchical Coder (also called the HCoder). By default, the Classic Coder is used for existing Visual State projects created with earlier versions of IAR Visual State, and the Hierarchical Coder is used for new projects. This setting is stored in the workspace file (`*.vnw`).

The Hierarchical Coder uses a more hierarchical approach for storing the data for the model. This means that it normally uses less constant data for models with many entry reactions, exit reactions, and history transitions. For small models, the resulting code might become slightly slower.

It is recommended that new models use the Hierarchical Coder. If you need to use RealLink or readable code, you must use the Classic Coder.

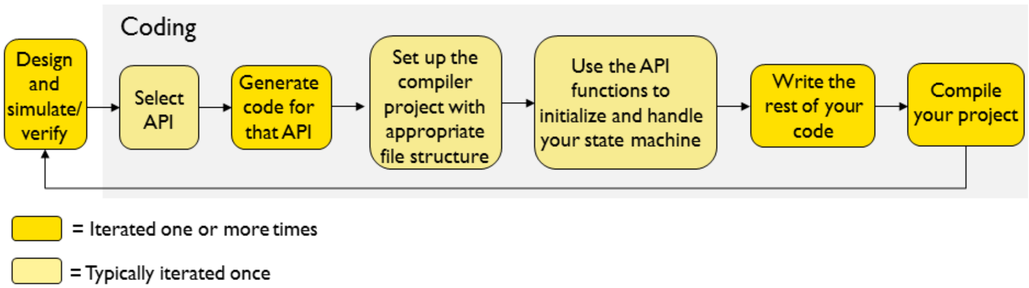
To select which code generator to use, open the Navigator, choose **Project>Options>Code Generation** and click the button **Switch Coder**.

CODE GENERATION USING THE VISUAL STATE CODERS

Based on the state machine models designed with the Visual State Designer, you can use one of the Visual State Coders to generate code automatically. The Coders generate code for a Visual State API (application programming interface). The generated code can be executed on any platform for which a standard compiler is available for the generated language, including small-scale microprocessor systems.

The Coder will generate code for one Visual State project at a time, including all systems and state machine models part of the project. All elements of a system are supported by the Coders. The generated code can also be used together with a real-time operating system. You start the code generation from the Navigator or from the command line.

This figure illustrates the workflow for generating code for your state machine model and integrating the generated code with your own source code using an API; typically the workflow is iterated many times, but note that the light-colored tasks are only performed in the first iteration:



A final application will consist of:

- The actual application using the state machine model(s)

  This includes all startup code and generic runtime library code as used by the particular target hardware and compiler.
- The API files for the execution engine
- A set of Coder-generated files, which consists of:
  - The state machine tables (for table-based code only)
  - 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 model.

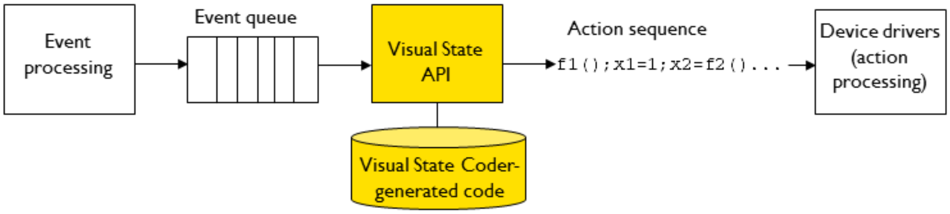
The Coders can generate a report file during code generation which contains this information for the project and the system:

- Coder options
- Model characteristics
- Generated statistics

- Information about the overall content of the generated code
- Number of errors and warnings detected during code generation.

THE VISUAL STATE APIS

The APIs are sets of files supplied with IAR Visual State and provide an interface between the Coder-generated code (highlighted in yellow) and your own code.



IAR Visual State includes three standard APIs as interfaces between Coder-generated code and your own code:

Adaptive API	This API can only be used with the Classic Coder. It is optimized for the data size of each system and has a copy of the API functions for each system. This makes it possible to generate smaller data (RAM use) per system at the expense of a tailored API runtime code (more ROM use). For projects with only one system, the Adaptive API is recommended.
Uniform API	This API can only be used with the Classic Coder. It uses one shared API for all systems and uses the same data sizes for all systems. Typically, this means that the data size might be larger and use extra RAM, but use less code because only one API is being used.
HCoder API	This is the only available API if you use the Hierarchical Coder. It uses one shared API for all systems and uses the same data sizes for all systems. Typically, this means that the data size might be larger and use extra RAM, but use less code because only one API is being used.

If you generate code using the Classic Coder, you should carefully consider which API to use:

Visual State API	Number of systems	Type of code	Language	RealLink	C-SPYLink
Adaptive	1	Table-based	C	Yes	Yes
	2 or more	Table-based	C	—	Yes
	1	Readable	C	—	Yes
	2 or more	Readable	C	—	Yes
	1	Readable	C#	—	—
	2 or more	Readable	C#	—	—
	1	Readable	Java	—	—
	2 or more	Readable	Java	—	—
	1	Table-based	C++	—	—
	2 or more	Table-based	C++	—	—
Uniform	1	Table-based	C	Yes	—
	2 or more	Table-based	C	Yes*	—
HCoder	1	Table-based	C	—	Yes
	2 or more	Table-based	C	—	Yes
	1	Table-based	C++	—	—
	2 or more	Table-based	C++	—	—

Table 25: Overview of the Visual State APIs

*) For the Uniform API, all systems must be run in the same task for RealLink to be supported.

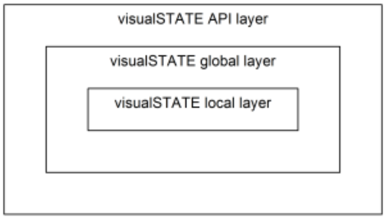
The data used by the API functions is formed in arrays and structures. The project is mapped in such a way that the computation of array component addresses is as simple and efficient as possible. This type of data representation ensures very low memory consumption, and compact and fast code.

Standard C conformance

The data structures and functions of all three APIs and the generated code conform to the ISO 9899:1990 standard (including all technical corrigenda and addenda), also known as C94, C90, C89, and ANSI C. In this guide, this standard is referred to as *Standard C*. For C++, the Adaptive and HCoder APIs conform to the ISO/IEC 14882:1998.

BRIEFLY ABOUT THE GENERATED CODE LAYERS

Visual State code is organized in different layers:



The Coder will generate the complete code for the API layer, the global layer, and the local layer.

The Visual State API layer

The API layer is the functions used for accessing the state machine engine and model during runtime. The API files are generated at the same time as the code is generated for the global and local layers.

The Visual State global layer

The global layer contains what you could call *external logic*. It is external in the sense that your code that uses the API can access the data in some way, for example by calling an API function. The global layer includes events, constants, enumerations, external variables, action expressions, and element explanations.

The Visual State local layer

The local layer contains the logic that is used internally in the model. Thus, it cannot be seen by your code that interfaces to the model. The local layer includes transitions, guard expressions, internal variables, and signals.

SIZE OF GENERATED TABLE-BASED CODE

A Visual State implemented application consists of the actual application using the state machine model, the API files, the generated code, and the action functions. All these parts are combined and give the footprint of the complete application. IAR Visual State only determines the size of the API and the generated code; you have full control of the other parts, which are more or less independent of the implementation of the state machine model. A typical Visual State application uses a limited set of the API functions to insert stimuli into the state machine and process input.

For the table-based API variants, the tables are generated in a way that is extremely compact, but which requires a runtime execution engine. This is common to all table-driven solutions and not limited to state machines.

The execution engine represents a fixed overhead in terms of code size. However, this overhead is small when used with a modern compiler. Because the code generated from the model is so tight, the advantage over hand-coding the model is apparent even for small state machines.

By default, the Coders will optimize for size. See also *Tailoring data types for a specific compiler*, page 463.

Tests for code size overhead

To measure the *minimum size* of the API code, you can create a minimal state machine and compile it. Typically, the state machine model should consist of an initial state, a simple state, and a default transition which contains an assignment to an externally defined variable. The API functions used for this model are the ones typically used by a Visual State application. (Most other functions available in the API are for advanced use and enable very fine-grained control of the state machine for debugging purposes.)

SIZE OF GENERATED READABLE CODE

The size of 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 readable code, this test code is not generated for transitions where it is trivial for the Coder to detect that there can be no transition contradictions. To turn off the generation of contradiction test code, see *-omitcontradictiontests*, page 729.

Moreover, readable code is much more dependent on the target compiler 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: Only the Classic Coder can generate readable code.

Generating code using a Coder and an API

What do you want to do:

- *Tailoring data types for a specific compiler*, page 463

See also:

- *Getting started generating code for the Adaptive API*, page 572
- *Getting started generating code for the Uniform API*, page 588

TAILORING DATA TYPES FOR A SPECIFIC COMPILER

The Coders will optimize for size. For the most efficient code size, consider these guidelines:

- The Coders will optimize the size of the API type definition, which can be 8 bits, 16 bits, or 32 bits. Use the Coder option `-D` to specify the width for all API type definitions, which are defined in *`System`*`.h` for the Adaptive API and in *`ProjectName`*`.h` for the HCoder and Uniform APIs.
- The transition rule data format is used for storing transition rules in the local code layer. Each transition consists of one rule data header word and one rule data element for each element of the transition rule. For the Classic Coder, the command line option `-rdfm` determines the rule data format to be used. For a list of transition rule data formats, see *Transition rule data format*, page 699.

HCoder API code generation

- Introduction to the HCoder API code generation
- Using the HCoder API

Before you read about HCoder API code generation, you should be familiar with code generation in general. See *Code generation*, page 457.

Introduction to the HCoder API code generation

Learn more about:

- *Briefly about HCoder API code generation*, page 465
- *API table-based code with C++*, page 466
- *API code*, page 467
- *Using the HCoder API for table-based code and C++*, page 467

BRIEFLY ABOUT HCODER API CODE GENERATION

The HCoder API will be generated in two files: `project.c` and `project.h`, where `project` reflects the name of your project file. Code for the HCoder API can only be generated by the Hierarchical Coder.

Most of the functions in the HCoder API have VS as prefix, and they take a pointer to a system context as a parameter to determine the system to operate on (if there are more than one system). This means that the API can operate on projects that contain multiple systems.

For information about the functions, see *Descriptions of the HCoder API functions*, page 474.

Projects with multiple systems and reentrancy

Because all functions are reentrant and are passed with a system context as a parameter, multiple operating system tasks may operate on different systems at the same time. Because of the principle of reentrancy, simultaneous calls made to the same API function will not cause problems as long as none of the simultaneous calls use the same system contexts as parameters to the function in question. Likewise, simultaneous calls to different API functions are supported. In general, simultaneous calls to API functions with different system contexts are supported. For example, event deductions may be in progress in different operating system tasks at the same time, executing the `VSDeduct` function.

This is an example of how system contexts are used; a system object pointer variable is defined for each system:

```
VSSystemObject* pSystemObject;
```

The system object pointer is assigned by calling the initialization function:

```
VSInitAll(&vssc_System, &pSystemObject);
```

The system object pointer used in an event deduction (macrostep):

```
VSTriggerType eventNo;
VC_RC cc;

. . .
cc = VSDeduct(pSystemObject, EventNo);
if (cc != VSRC_OK)
  exit (cc);
```

Reentrancy of API functions depends on the compiler you use. Thus, the compiler used for compilation of API source files must also support reentrancy, because some of the API functions use local stack variables. If the compiler does not support reentrancy, local stack variables might be stored in fixed memory locations, and different operating system tasks controlling different systems might access the same variable space simultaneously which will result in unpredictable behavior.

API TABLE-BASED CODE WITH C++

The Hierarchical Coder can generate C++ code for the HCoder API. The generated C++ files conform to the Embedded C++ standard.

C++ code generated for the HCoder API uses C++ to expose its external interface, but uses C internally to keep the generated code compact and efficient. Do not call the C code directly when C++ has been generated, because this can cause undefined behavior and crashes.

Generating C++ code has the following advantages:

- User-written code that interfaces to the generated code can interact with a class that uses C++ language features such as the keyword `private` to protect its members from accidental and/or prohibited access.
- To many developers, exposing a C++ interface is more elegant than exposing a C interface.

- In your user-written code you can create any number of instances of the Visual State system, and the instances can be allocated statically or dynamically at the same time. This feature is not available in the API when generating C code. In addition, instances do not share any internal data memory (do not include external variables) and therefore it is easier to enable thread safety in your application.

The performance of C++ code generated for the API is about similar to the performance of C code generated for the API.

File structure

The file structure to be used in your compiler project—for example, in the IAR Embedded Workbench IDE—is the same for code with and without C++ support.

Using the default Hierarchical Coder options, the generated C source files have the filename extension `c`, and the generated C++ source files have the filename extension `cpp`. You can change these extensions in the **Hierarchical Coder Options** dialog box.

API CODE

During the code generation phase, these sets of files are generated:

- Project-specific API files, typically *`project`*`.c` and *`project`*`.h`, where *`project`* reflects the name of your project file
- System-specific files: *`System1`*`.c`, *`System1`*`.h`, *`System2`*`.c`, *`System2`*`.h`, etc, where *`SystemN`* reflect the names of the Visual State systems.

USING THE HCODER API FOR TABLE-BASED CODE AND C++

The Hierarchical Coder does not instantiate objects of the generated system class. Therefore, you must instantiate objects of the system class in your own files (user-written code).

In contrast to a C API application, any number of objects of the system class can be instantiated, just as is the case for ordinary classes.

When generating C++ code, you must interface to member functions of the generated system class instead of global functions. For every API function that you must call for a C application, you must call a corresponding member function (having the same name) of the generated class. The action functions must be implemented in a user-written class, inheriting from the system class (or project class, if shared action functions are used).

Instances in C++ API code

Each Visual State system consists of one or more instances with exactly one instance being active at any point in time, see *Reuse of design using system instances*, page 126. Such instances are called *internal instances* and they have these characteristics:

- The number of internal instances is fixed for the system at the time of code generation. You can specify the number of internal instances in the Designer in the **Edit Systems** dialog box. See *Creating multiple system instances*, page 235.
- Only one internal instance may be active at a time because internal instances share internal data memory.

Internal instances should not be mistaken for instances (objects) of the generated class, which are called *external instances* and can be instantiated any number of times, either statically, on the stack, or in the heap.

Both types of instances may be referred to as just instances when the type of instance clearly appears from the context.

Internal variables in C++ API code

Internal variables are part of the generated class as private member variables. Consequently they can only be initialized by an initialization function.

External variables in C++ API code

External variables are not part of the generated class, but are generated as statically allocated variables, in the same way as for a C application. Therefore, all external instances of the generated class share the same set of external variables.

If two external instances manipulate an external variable from two different threads, you must synchronize the access to that variable.

Constants in C++ API code

Constants are not part of the generated class, but are part of the namespace given for the system or project (depending on the scope).

Enumerations in C++ API code

Enumerations are not part of the generated class, but are part of the namespace given for the system or project (depending on the scope). They are always generated in a separate header file for each enumeration.

Signals in C++ API code

Signals are handled internally, in the same way as for a C application. Note that every external instance has its own signal queue (assuming dynamic allocation), while internal instances share a single signal queue.

Event parameters in C++ API code

Event parameters are handled in the same way as for a C application. The Hierarchical Coder will always generate a member function `VSDeduct` for the generated class, independently of the existence of event parameters. Note that variable argument parameters are not supported in C++.

Using the HCoder API

What do you want to do:

- *Setting up the file structure for the HCoder API*, page 469

See also:

- *Introduction to code generation, the Coders, and the APIs*, page 457
- *Introduction to the HCoder API code generation*, page 465
- *HCoder API reference information*, page 471
- *Hierarchical Coder command line options*, page 517, for information about how to start code generation from the command line

SETTING UP THE FILE STRUCTURE FOR THE HCODER API

1. Include these header files in all your source files:
   - The HCoder API header file `project.h`.
   - The Hierarchical Coder-generated header file for the specific system, in other words `system.h`.
2. Write code that interfaces to the API:
   - Call `VSInitAll`, once for each system.
   - Call `VSDeduct` to process events.
3. Include the following source files in a make file:
   - The HCoder API source file `project.c`.
   - All Coder-generated system source files.
   - Your source files.

4 Add your compiler and linker commands to the make file.

HCoder API reference information

- HCoder API source files
- Summary of the HCoder API functions
- Descriptions of the HCoder API functions
- HCoder API return codes

HCoder API source files

The HCoder API will be generated in two files: *`project`*`.c` and *`project`*`.h`, where *`project`* reflects the name of your project. Most of the functions in the API have `VS` as a prefix and they take a pointer to a system object as a parameter to determine the system to operate on (this parameter is not used if there is only one system). This means that the API can operate on projects that contain multiple systems.

Unless otherwise stated, portability of the HCoder API is Standard C compliant.

What do you want to do:

- *HCoder-generated source files for the API*, page 471

HCODER-GENERATED SOURCE FILES FOR THE API

During the code generation phase, these sets of files are generated:

- Project-specific files
- System-specific files

These are the project-specific files:

`project``.h`	Contains the declarations of the API functions, macros and types; *`project`* reflects the name of your project (or project class, for C++).
`project``.c`	Contains internal types and the implementation of the API functions; *`project`* reflects the name of your project (or project class, for C++).

These are the system-specific files (for each system):

system.c	Contains the declarations of system-specific variables, macros and functions; *system* reflects the name of your system (or system class, for C++).
system.h	Contains definitions of internal variables and functions; *system* reflects the name of your system (or system class, for C++).

Summary of the HCoder API functions

This table summarizes the HCoder API functions:

HCoder API function	Description
VSActiveState*	Get the active state in a machine.
VSDeduct*	Deduces an event.
VSDelete*	Deallocates a system object.
VSEnterState†	A user-defined function is called when a state is entered.
VSEventExpl*	Gets an event explanation.
VSEventName*	Gets an event name.
VSGetSystemObjectSize*	Gets the system object size.
VSInitAll*	Initializes a system object and all its internal variables.
VSInquiry*	Inquires an event.
VSLeaveState†	A user-defined function is called when a state exits.
VSMachineExpl*	Gets a machine explanation.
VSMachineName*	Gets a machine name.
VSNew*	Allocates and initializes a system object.
VSNofEventParameters*	Gets the number of event parameters.
VSNofEvents*	Gets the number of events in the scope of the system.
VSNofInstances*	Gets the number of internal instances.
VSNofMachines*	Gets the number of state machines.
VSNofStates*	Gets the number of states.
VSNofVariables*	Gets the number of variables.
VSParentMachine*	Gets the parent machine of a state.
VSParentState*	Gets the parent state of a machine.
VSReinitialize*	Reinitializes the active internal instance.
VSSetInstance*	Sets the internal instance within the system object.
VSStateName*	Gets a state name.
VSSymbolicStateName*	Behaves identically to VSStateName.
VSSymbolicVariableName*	Gets a symbolic variable name.
VSTopMachine*	Gets a top machine.
VSVariableValue*	Gets a variable value as a string.

Table 26: Summary of the HCoder API functions

* The function is only generated if the appropriate Hierarchical Coder option has been

enabled. For more information, see the individual function.

† The function is only enabled and called if specific options have been enabled. For more information, see the individual function.

Descriptions of the HCoder API functions

The following pages give detailed reference information about each HCoder API function. The C++ API functions and the C API functions are identical, except that system object and system class parameters are never needed in the C++ API. The C++ API functions are declared in the *system*.h file and are always called by calling a system object member function.

VSActiveState

Syntax

```
VSRC VSActiveState(VSMachineType const machineNo, VSStateType *
                   const pStateNo);
VSRC VSActiveState(VSSystemObject * const pSystemObject,
                   VSMachineType const machineNo, VSStateType *
                   const pStateNo);
```

Declared in

`project.h`

Description

Gets the active state of a machine. This function is enabled on demand. The function returns the active state of the specified machine in the parameter `pStateNo`.

Parameters

`pSystemObject`	A pointer to a system object.
`machineNo`	The number of the machine for which to determine the active state.
`pStateNo`	A pointer to a state number. When the function returns, the value of the variable will be the number of the active state. If the machine is not active, `pStateNo` will be set to `VSStateUndefined`.

Return value

VSRC_OK, page 492

VSRC_RangeError, page 492

VSDeduct

Syntax

```
VSRC VSDeduct(VSTriggerType eventNoArg, ...);
VSRC VSDeduct(VSSystemObject * const pSystemObject,
              VSTriggerType eventNoArg);
```

Declared in

`project.h`

Description

Deduces an event. This function is always enabled and performs a macrostep for the supplied event for the specified system object.

Parameters

`pSystemObject`	A pointer to a system object.
`eventNoArg`	The event number to be deduced (declared in *system*.h or *project*.h).

Return value

VSRC_OK, page 492

VSRC_RangeError, page 492

VSRC_Conflict, page 491

VSRC_SignalQueueOverflow, page 492

Example

```
/* Event E_Event1 without parameters */
if (VSDeduct(E_Event1) != VSRC_OK)
  ErrorHandling ();
/*
* Event E_Event2 with two parameters:
* Argument 1: unsigned int Par1
* Argument 2: unsigned short Par2
*/
if (VSDeduct(E_Event2, Par1, Par2) != VSRC_OK)
  ErrorHandling ();
void Task(void)
{
  VCRC cc;
  VSTriggerType eventNo = VSStartEvent;
  /* Initialize the VS System. */
  cc = VSInitAll();
  if (cc != VCRC_OK)
    handleError(cc);
  /* do forever */
  while (1)
  {
    cc = VSDeduct(eventNo);
    if (cc != VCRC_OK)
      handleError(cc);
  }
}
/* Multiple systems variant. */
void Task(void)
{
  VSSystemObject* pSystemObject;
  VSTriggerType eventNo = VSStartEvent;
  VCRC cc;
  /* vssc_System1 is in the generated System1.h file. */
  cc = VSInitAll(&vssc_System1, &pSystemObject);
  if (cc != VCRC_OK)
    handleError(cc);

  while (1)
  {
    cc = VSDeduct(pSystemObject, eventNo);
    if (cc != VCRC_OK)
      handleError(cc);
  }
}
```

VSDelete

Syntax

C: `void VSDelete(VSSystemObject * const pSystemObject);`

C++: Not applicable. Called when a destructor of a system object is executed.

Declared in

project.h

Description

Deallocates a system object. This function is enabled when the project contains multiple systems that will be dynamically allocated. The function deallocates a system object allocated by `VSNew`. The function is analogous to the `delete` operator in C++ in that it deallocates memory for a previously allocated system object. The function should be called for all system objects allocated with `VSNew`.

Parameter

pSystemObject	A pointer to a system object.

Return value

None.

Example

```
VSDelete(pObject);
```

VS*Project*EnterState

Syntax

```
void VSProjectEnterState(VSStateType const stateNo);
void VSProjectEnterState(VSSystemObject * const pSystemObject,
                         VSStateType const stateNo);
```

Declared in

project.h

Description

A user-defined function that is called when a state is entered. *Project* in VS*Project*EnterState is the name of the active project.

Parameters

pSystemObject	A pointer to a system object.
StateNo	The name of the state that was entered.

Return value

None.

VSEventExpl

Syntax

```
void VSEventExpl(VSTriggerType eventNo, VS_CHAR const * *
                 ppExpl);
void VSEventExpl(VSSystemClass const * const pSystemClass,
                 VSTriggerType eventNo, VS_CHAR const * *
                 ppExpl);
```

Declared in

project.h

Description

Gets an event explanation. This function is enabled on demand. The function retrieves the explanation of the specified event. If the event is defined at project level, the function will also return the explanation for it.

Parameters

pSystemClass	A pointer to a system class.
eventNo	The number of the event for which to return the explanation. The maximum allowable value for this parameter is the value returned by the API function VSNofEvents minus 1.
ppExpl	A pointer to an explanation. When the function returns, the variable will point to the explanation for the specified event.

Return value

None.

Example

```
VSEventExpl(&vssc_System1, eventNo, &pExpl);
```

VSEventName

Syntax

```
VSRC VSEventName(VSTriggerType eventNo, VS_CHAR const * *
                 ppName);
VSRC VSEventName(VSSystemClass const * const pSystemClass,
                 VSTriggerType eventNo, VS_CHAR const * *
                 ppName);
```

Declared in

project.h

Description

Gets an event name. This function is enabled on demand. The function retrieves the name of the specified event. If events are defined at project level, the function will also return names for such events.

Parameters

`pSystemClass`	A pointer to a system class.
`eventNo`	The number of the event for which to return the name. The maximum allowable value for this parameter is the value returned by the API function `VSNofEvents` minus 1.
`ppName`	A pointer to a name. When the function returns, the variable will point to the name for the specified event.

Return value

VSRC_OK, page 492

VSRC_RangeError, page 492

Example

```
VSRC rc = VSEventName(&vssc_System1, &pName);
```

VSGetSystemObjectSize

Syntax

```
VS_UINT VSGetSystemObjectSize();
VS_UINT VSGetSystemObjectSize(VSSystemClass const * const
                              pSystemClass);
```

Declared in

project.h

Description

Gets a system object size. This function is enabled on demand. The function returns the size in bytes of system objects of the specified system class.

Parameters

`pSystemClass`	A pointer to a system class.

Return value

The size of the system object.

VSInitAll

Syntax

```
VSRC VSInitAll(void);
VSRC VSInitAll(VSSystemClass const * const pSystemClass,
               VSSystemObject * * const pSystemObject);
```

Declared in

project.h

Description

Initializes a system object. This function is enabled when system objects are statically allocated. If the project contains a single system, the function initializes the single statically allocated system object. If the project contains multiple systems, the function initializes the statically allocated system object for the specified system class and returns a pointer to that system object. The function should only be called once for each system class; multiple calls to this function with the same system class cause undefined behavior.

Parameters

`pSystemClass`	A pointer to a system class.
`pSystemObject`	A pointer to a pointer to a system object. When the function returns, the variable will contain a pointer pointing to an initialized system object.

Return value

VSRC_OK, page 492

VSRC_Conflict, page 491

VSRC_SignalQueueOverflow, page 492

Example

See *VSDeduct*, page 475.

VSInquiry

Syntax

```
VSRC VSInquiry(VSTriggerType eventNo, ...);
VSRC VSInquiry(VSSystemObject * const pSystemObject,
               VSTriggerType eventNo, ...);
```

Declared in

`project``.h`

Description

Inquires an event. This function is enabled on demand. The function performs an inquiry for a specified event and returns `EventActive` if the event will trigger a trans reaction. The function has the same interface as the `VSDeduct` function.

Parameters

`pSystemObject`	A pointer to a system object.
`eventNo`	The event number to be inquired.

Return value

VSRC_OK, page 492

VSRC_EventActive, page 491

Example

See *VSDeduct*, page 475.

VS*Project*LeaveState

Syntax

```
void VSProjectLeaveState(VSStateType const stateNo);
void VSProjectLeaveState(VSSystemObject * const pSystemObject,
                         VSStateType const stateNo);
```

Declared in

project.h

Description

A user-defined function that is called when a state is exited. *Project* in VS*Project*LeaveState is the name of the active project.

Parameters

pSystemObject	A pointer to a system object.
StateNo	The name of the state that was exited.

Return value

None.

VSMachineExpl

Syntax

```
void VSMachineExpl(VSMachineType machineNo, VS_CHAR const * *
                   ppExpl);
void VSMachineExpl(VSSystemClass const * const pSystemClass,
                   VSMachineType machineNo, VS_CHAR const * *
                   ppExpl);
```

Declared in

project.h

Description

This function is enabled on demand. The function retrieves the explanation of the specified machine.

Parameters

pSystemClass	A pointer to a system class.
machineNo	The number of the machine for which to return the explanation. The maximum allowable value for this parameter is the value returned by the API function VSNofMachines minus 1.
ppExpl	A pointer to an explanation. When the function returns, the variable will point to the explanation for the specified machine.

Return value

None.

VSMachineName

Syntax

```
VSRC VSMachineName(VSMachineType machineNo, VS_CHAR const * *
                   ppName);
VSRC VSMachineName(VSSystemClass const * const pSystemClass,
                   VSMachineType machineNo, VS_CHAR const * *
                   ppName);
```

Declared in

project.h

Description

Gets a machine name. This function is enabled on demand. The function retrieves the name of the specified machine.

Parameters

pSystemClass	A pointer to a system class.
machineNo	The number of the machine for which to return the name. The maximum allowable value for this parameter is the value returned by the API function VSNofMachines minus 1.
ppName	A pointer to a name. When the function returns, the variable will point to the name of the specified event.

Return value

VSRC_OK, page 492

VSRC_RangeError, page 492

Example

```
VSRC rc = VSMachineName(machineNo, &pName);
```

VSNew

Syntax

C: `VSRC VSNew(VSSystemObject * * const pSystemObject);`

C++: Not applicable. Called internally when `VSInitAll` is called.

Declared in

project.h

Description

Allocates and initializes a system object. This function is enabled when system objects are dynamically allocated. The function returns a pointer to an initialized system object for the system associated with the specified system class parameter. The behavior of the function is analogous to the `new` operator in C++ in that it allocates memory for a new system object and initializes it. When the system object is no longer needed, call

VSDelete to deallocate it. The function should be called whenever a new system object is needed.

Parameter

pSystemObject	A pointer to a pointer to a system object. When the function returns, the variable will contain a pointer pointing to an initialized system object.

Return value

VSRC_OK, page 492

VSRC_CannotAllocateMemory, page 491

Example

```
VSSystemObject* pObject;
VSRC cc = VSNew(&pObject);
if (cc != VSRC_OK)
  handleError(cc);
```

VSNofEventParameters

Syntax

```
VSRC VSNofEventParameters(VSTriggerType const eventNo,
              VSEventParameterType * const pNofEventParameters);
```

Declared in

project.h

Description

Gets the number of event parameters. This function is enabled on demand.

Parameters

eventNo	The event for which the number of event parameters is requested.
pNofEventParameters	A pointer to the number of event parameters. When the function returns, the value of the variable will be the number of event parameters for the specified event.

Return value

VSRC_OK, page 492

VSRC_RangeError, page 492

Example

```
VSEventParameterType noOfEventParams;
VSNofEventParameters(Event1, &noOfEventParams);
```

VSNofEvents

Syntax

```
VSTriggerType VSNofEvents();
```

Declared in

project.h

Description

Gets the number of events in the scope for the system. This function is enabled on demand. If events are defined at project level, the returned value will include such events.

Return value

The number of events.

Example

```
VSTriggerType noOfEvents = VSNofEvents();
```

VSNofInstances

Syntax

```
VSInstanceType VSNofInstances();
VSInstanceType VSNofInstances(VSSystemClass const * const
                              pSystemClass);
```

Declared in

project.h

Description

Gets the number of internal instances. This function is enabled on demand.

Parameters

pSystemClass	A pointer to a system class.

Return value

The number of instances.

Example

```
VSInstanceType noOfInstances = VSNofInstances();
```

VSNofMachines

Syntax

```
VSMachineType VSNofMachines();
VSMachineType VSNofMachines(VSSystemClass const * const
                            pSystemClass);
```

Declared in

project.h

Description

Gets the number of state machines. This function is enabled on demand.

Parameters

`pSystemClass`	A pointer to a system class.

Return value

The number of machines.

Example

```
VSMachineType nofMAchines = VSNofMachines();
```

VSNofStates

Syntax

```
VSStateType VSNofStates();
VSStateType VSNofStates(VSSystemClass const * const
                        pSystemClass);
```

Declared in

project.h

Description

Gets the number of states. This function is enabled on demand.

Parameters

`pSystemClass`	A pointer to a system class.

Return value

The number of states.

Example

```
VSStateType nofStates = VSNofStates();
```

VSNofVariables

Syntax

```
VSVariableType VSNofVariables();
VSVariableType VSNofVariables(VSSystemClass const * const
                              pSystemClass);
```

Declared in

project.h

Description

Gets the number of variables. This function is enabled on demand.

Parameters

`pSystemClass`	A pointer to a system class.

Return value

The number of variables.

Example

```
VSVariableType nofVariables = VSNofVariables();
```

VSParentMachine

Syntax

```
VSRC VSParentMachine(VSStateType const stateNo, VSMachineType *
                     const pMachineNo);
VSRC VSParentMachine(VSSystemClass const * const pSystemClass,
                     VSStateType const stateNo, VSMachineType *
                     const pMachineNo);
```

Declared in

`project.h`

Description

Gets the parent machine of a state. This function is enabled on demand. The function returns the parent machine of the specified state (a state always has a parent machine).

Parameters

`pSystemClass`	A pointer to a system class.
`stateNo`	The number of the state for which to determine the parent machine.
`pMachineNo`	A pointer to a machine number. When the function returns, the value of the variable will be the number of the parent machine.

Return value

VSRC_OK, page 492

VSRC_RangeError, page 492

VSParentState

Syntax

```
VSRC VSParentState(VSMachineType const machineNo, VSStateType *
                   const pStateNo);
VSRC VSParentState(VSSystemClass const * const pSystemClass,
                   VSMachineType const machineNo, VSStateType *
                   const pStateNo);
```

Declared in

`project.h`

Description

Gets the parent state of a machine. This function is enabled on demand. The function returns the parent state of the specified machine in the parameter `pStateNo`.

Parameters

`pSystemClass`	A pointer to a system class.
`machineNo`	The number of the machine for which to determine the parent state.

`pStateNo`	A pointer to a state number. When the function returns, the value of the variable will be the number of the parent state. If the specified machine is the top machine, `pStateNo` will be set to `VSStateUndefined`.

Return value

VSRC_OK, page 492

VSRC_RangeError, page 492

VSReinitialize

Syntax

```
VSRC VSReinitialize(VSSystemObject * const pSystemObject);
```

Declared in

project.h

Description

Reinitializes the active internal instance. This function is enabled on demand. The function reinitializes the active internal instance within the specified system object. The state of the internal instance is the same as after a call to `VSInitAll` or `VSNew`. The system object must be initialized; otherwise the behavior of the function is undefined. The function is useful for reuse of an internal instance in several cases, for example when an internal instance has failed with a detected conflict or signal queue overflow. or when a set of internal instances within a system object is used as a pool available for reuse in user-written code.

Parameter

`pSystemObject`	The address of the system object pointer that is initialized.

Return value

VSRC_OK, page 492

VSRC_CannotAllocateMemory, page 491

VSSetInstance

Syntax

```
VSRC VSSetInstance(VSInstanceType instanceNo);
VSRC VSSetInstance(VSSystemObject * const pSystemObject,
                   VSInstanceType instanceNo);
```

Declared in

project.h

Description

Sets the internal instance within the system object. This function is enabled when at least one system object contains multiple internal instances. The function makes a specific internal instance active. Subsequent calls to functions that operate on an internal instance will operate on this internal instance.

Parameters

`pSystemObject`	A pointer to a system object.
`instanceNo`	The internal instance number to make active.

Return value

VSRC_OK, page 492

VSRC_RangeError, page 492

Example

```
/* Multiple instances. */
void Task(VSInstanceType inst)
{
  VSSystemObject* pSystemObject;
  VSTriggerType eventNo = VSStartEvent;
  VCRC cc;
  /* vssc_System1 is in the generated System1.h file. */
  cc = VSInitAll(&vssc_System1, &pSystemObject);
  if (cc != VCRC_OK)
    handleError(cc);

  cc = VSSetInstance(pSystemObject, inst);
  if (cc != VCRC_OK)
    handleError(cc);

  while (1)
  {
    cc = VSDeduct(pSystemObject, eventNo);
    if (cc != VCRC_OK)
      handleError(cc);
  }
}
```

VSStateName

Syntax

```
VSRC VSStateName (VSStateType stateNo, VS_CHAR const * *
                  ppName);
VSRC VSStateName (VSSystemClass const * const pSystemClass,
                  VSStateType stateNo, VS_CHARconst * * ppName);
```

Declared in

project.h

Description

Gets a state name. This function is enabled on demand.

Parameters

`pSystemClass`	A pointer to a system class.
`stateNo`	The number of the state for which to return the state name. The maximum allowable value for this parameter is the value returned by the API function `VSNofStates` minus 1.
`ppName`	A pointer to a state name. When the function returns, the variable will point to the state name of the specified state.

Return value

VSRC_OK, page 492

VSRC_RangeError, page 492

VSSymbolicVariableName

Syntax

```
VSRC VSSymbolicVariableName(VSVariableType variableNo, VS_CHAR
                            const * * ppName);
VSRC VSSymbolicVariableName(VSSystemClass const * const
                            pSystemClass, VSVariableType
                            variableNo, VS_CHAR const * *
                            ppName);
```

Declared in

project.h

Description

Gets a symbolic variable name. This function is enabled on demand. The function returns the symbolic variable name for the specified variable number as a string. The variable for which to return the name is specified as a variable number (the range of allowed values for this number can be obtained via another API function). Instead of using plain numbers, it is recommended to use symbolic variable names that map to variable numbers (symbolic variable names are enabled by setting the appropriate option).

Parameters

`pSystemClass`	A pointer to a system class.
`variableNo`	The number of the variable for which to return the symbolic variable name. The maximum allowable value for this parameter is the value returned by the API function `VSNofVariables` minus 1.
`ppName`	A pointer to a symbolic variable name. When the function returns, the variable will point to the symbolic variable name of the specified variable.

Return value

VSRC_OK, page 492

VSRC_RangeError, page 492

VSTopMachine

Syntax

```
VSMachineType VSTopMachine();
VSMachineType VSTopMachine(VSSystemClass const * const
                           pSystemClass);
```

Declared in

project.h

Description

Gets the top machine. This function is enabled on demand. The function returns the top machine of the hierarchy for the system.

Parameters

pSystemClass	A pointer to a system class.

Return value

The number of the top machine.

VSVariableValue

Syntax

```
VSRC VSVariableValue(VSVariableType const variableNo, VS_CHAR *
                     const pValue);
VSRC VSVariableValue(VSSystemObject * const pSystemObject,
                     VSVariableType const variableNo, VS_CHAR *
                     const pValue);
```

Declared in

project.h

Description

Gets a variable value as a string. This function is enabled on demand. The function returns the value of a specified external or internal variable as a string. For arrays the function will return the value for a single element in the variable array. The function will return values for variables that are in the scope of the specified system object, in other words external and internal variables defined at top state level and external variables defined at project level. The variable for which to return a value is specified as a variable number (the range of allowed values for this number can be obtained via another API function). Instead of using plain numbers, you should use symbolic variable names that map to variable numbers (symbolic variable names are enabled by setting the appropriate option). Symbolic variable names can also be obtained as strings by enabling the appropriate option.

Parameters

`pSystemObject`	A pointer to a system object.
`variableNo`	The number of the variable (use the symbolic name) for which to determine the value.
`pValue`	A pointer value as a string. When the function returns, the value of the variable will be represented as a string. The character buffer must be large enough to hold the value.

Return value

VSRC_OK, page 492

VSRC_RangeError, page 492

HCoder API return codes

The following pages give detailed reference information about each HCoder API return code.

VSRC_CannotAllocateMemory

Return code `VSRC_CannotAllocateMemory`

Description The function failed to dynamically allocate a system object.

Solution

- Free some memory on the host computer
- Use a large data memory model.

VSRC_Conflict

Return code `VSRC_Conflict`

Description A conflict or contradiction has been detected between two states in a state machine.

Solution Check the system with the Validator or the Verificator and change the design as needed.

VSRC_EventActive

Return code `VSRC_EventActive`

Description Sending the event to a `Deduct` function will trigger a trans reaction.

Solution Not applicable.

VSRC_OK

Return code `VSRC_OK`

Description The function performed successfully, unless it was an `Inquiry` function. `Inquiry` functions are expected to return `VSRC_EventActive` (`VSRC_OK` means that the event is not active).

Solution Not applicable.

VSRC_RangeError

Return code `VSRC_RangeError`

Description An in parameter was sent in that was too large.

Solution Check the code that calls the method returning the error code. The supplied argument is out of range.

VSRC_SignalQueueOverflow

Return code `VSRC_SignalQueueOverflow`

Description The signal queue is full.

Solution Increase the maximum signal queue size in your system or change the design.

The Visual State Hierarchical Coder

- Introduction to the Visual State Hierarchical Coder
- Graphical environment for the Hierarchical Coder
- Type identifiers
- Transition rule data format

Introduction to the Visual State Hierarchical Coder

Learn more about:

- *Briefly about the Visual State Hierarchical Coder*, page 493

BRIEFLY ABOUT THE VISUAL STATE HIERARCHICAL CODER

There are two Visual State Coders to use for generating code from your state machine models for a specific API. For more information about code generation and the APIs, see *Code generation*, page 457.

Before you start the code generation, specify Coder options in the **Hierarchical Coder Options** dialog box. Start the code generation by choosing **Project>Code generate** in the Navigator.

For a description of the Visual State Classic Coder, see *The Visual State Classic Coder*, page 673.

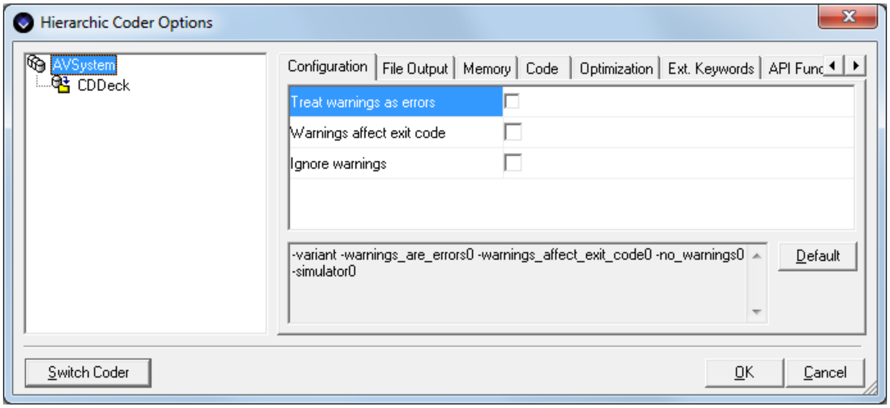
Graphical environment for the Hierarchical Coder

Reference information about:

- *Hierarchical Coder Options dialog box*, page 494

Hierarchical Coder Options dialog box

The **Hierarchical Coder Options** dialog box is available from the **Project** menu in the Navigator.



Use this dialog box to set options for code generation. Which options you can set depends on whether you are setting options on project level or on system level. Select either the project or a system in the pane to the left.

Use the **Switch Coder** button to switch from the Hierarchical Coder to the Classic Coder and back again.

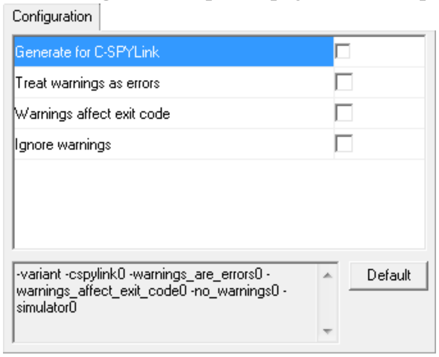
For a description of an option, right-click it or select it and press Shift+F1.

You can set options on these tabbed pages:

- *Hierarchical Coder Options dialog box : Configuration*, page 495
- *Hierarchical Coder Options dialog box : File Output*, page 496
- *Hierarchical Coder Options dialog box : Memory*, page 498
- *Hierarchical Coder Options dialog box : Code*, page 499
- *Hierarchical Coder Options dialog box : Optimization*, page 504
- *Hierarchical Coder Options dialog box : Extended Keywords*, page 508
- *Hierarchical Coder Options dialog box : API Functions*, page 510
- *Hierarchical Coder Options dialog box : C-SPYLink*, page 511
- *Hierarchical Coder Options dialog box : Names*, page 513

Hierarchical Coder Options dialog box : Configuration

The **Configuration** options page contains options for general configuration.



Use this page to make configuration settings for the Hierarchical Coder. The display area under the options shows the resulting command line for the code generation.

This options page can be accessed on both project level and system level. Exactly which options you can set depends on the level you are setting options on.

Generate for C-SPYLink

Choose whether to generate code to be debugged using C-SPYLink.

Treat warnings as error

Makes the Hierarchical Coder treat all warnings as errors. If the Coder encounters an error, no code is generated. This option can only be set on project level.

Warnings affect exit code

By default, the exit code is not affected by warnings, because only errors produce a non-zero exit code. With this option, warnings will also generate a non-zero exit code. This option can only be set on project level.

Ignore warnings

By default, the Hierarchical Coder issues warnings. Select this option to disable all warnings. This option can only be set on project level.

Exclude system from build

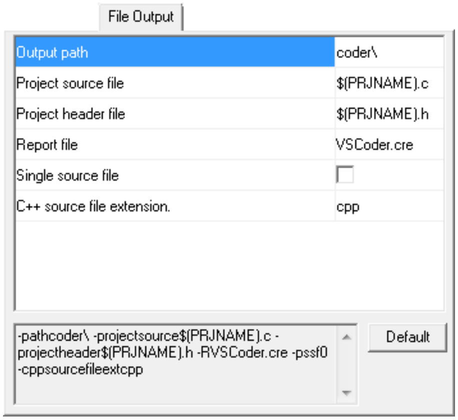
Determines whether the selected system will be part of the generated code or not. This option can only be set on system level.

Default

Restores the options to their default settings.

Hierarchical Coder Options dialog box : File Output

The **File Output** options page contains options for file output from code generation.



Use this page to make file output settings for the Hierarchical Coder. The display area under the options shows the resulting command line for the code generation.

This options page can be accessed on both project level and system level. Exactly which options you can set depends on the level you are setting options on.

Use project output path

Makes the Hierarchical Coder use the same output path for system files as the path specified for all project files. This option can only be set on system level.

Output path

Specify the output path for all generated project or system files, respectively. If the path does not exist, it is created. The path can be relative. This option can be set on both project level and system level.

System header file

Specify the name of the header file that contains system-level model declarations. The name used by default is `System.h`. This option can only be set on system level.

System source file

Specify the name of the source file that contains system-level model definitions. The name used by default is `System.c`. This option can only be set on system level.

Project source file

Specify the name of the source file that contains project-level model definitions. The name used by default is `Project.c`. This option can only be set on project level.

Project header file

Specify the name of the header file that contains project-level model declarations. The name used by default is `Project.h`. This option can only be set on project level.

Report file

Specify a name for a report file to contain information about the project, option settings, model characteristics, statistics, and a summary of the code generation. The name used by default is `VSCoder .h`. This option can only be set on project level.

Single source file

Merges all project and system source files into the main project source file. The header files remain separate. This option can only be set on project level.

C++ source file extension

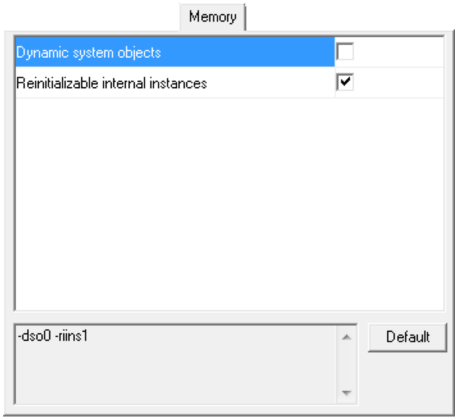
Type the filename extension that IAR Visual State shall use for generated C++ language source files. This option can only be set on project level.

Default

Restores the options to their default settings.

Hierarchical Coder Options dialog box : Memory

The **Memory** options page contains options for memory configuration.



Use this page to make memory settings for the Hierarchical Coder. The display area under the options shows the resulting command line for the code generation.

This options page can only be accessed on project level.

Dynamic system objects

Makes the Hierarchical Coder allocate systems objects dynamically instead of statically.

Reinitializable internal instances

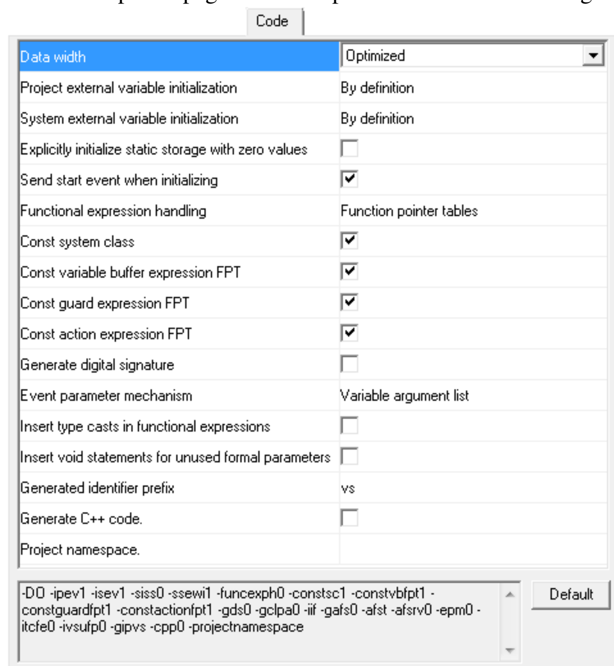
Specify whether the internal instances can be reinitialized or not. If this option is deselected, a reset is required to reach the initial state.

Default

Restores the options to their default settings.

Hierarchical Coder Options dialog box : Code

The **Code** options page contains options for the actual code generation.



Use this page to make code settings for the Hierarchical Coder. The display area under the options shows the resulting command line for the code generation.

This options page can be accessed on both project level and system level. Exactly which options you can set depends on the level you are setting options on.

Data width

Specify the width of internal data members. If you set the width to a smaller value than needed, the smallest possible value will be used. This option can only be set on project level.

Choose between:

Optimized

The smallest possible value will be used.

8-bit

Internal data members are 8-bit.

16-bit

Internal data members are 16-bit.

32-bit

Internal data members are 32-bit.

Project external variable initialization

Specify how to initialize project-external variables. This option can only be set on project level.

Choose between:

Never

Project-external variables are not initialized by the Hierarchical Coder. You must include initialization routines in your user-written application code.

By definition

Initializes project-external variables along with their definition.

With system objects

Initializes project-external variables when the system objects are initialized.

With internal instances

Initializes project-external variables when the internal instances are initialized.

System external variable initialization

Specify how to initialize system-external variables. This option can only be set on project level.

Choose between:

Never

System-external variables are not initialized by the Hierarchical Coder. You must include initialization routines in your user-written application code.

By definition

Initializes system-external variables along with their definition.

With system objects

Initializes system-external variables when the system objects are initialized.

With internal instances

Initializes system-external variables when the internal instances are initialized.

Explicitly initialize static storage with zero values

Initializes static storage with zero. If this option is deselected, static storage is left undefined unless initial values have been specified. This option can only be set on project level.

Send start event when initializing

Sends the start event automatically after `VSInitAll` has been executed. If this option is deselected, you must pass the start event into `VSDeduct` manually. This option can only be set on project level.

Functional expression handling

Specify how to handle functional expressions (guard expressions and action expressions). This option can only be set on project level.

Choose between:

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.

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 alternative settings efficiently.

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 can recognize the switch-case construct and convert it into a jump table, this might be the most efficient setting.

Const system class

Defines system class variables as `const` variables. This option should only be deselected in exceptional cases, for example, when the target controller has sufficient and fast RAM, and speed is of the highest importance. This option can only be set on project level.

Const variable buffer expression FPT

Defines the variable buffer expression function pointer table as a `const` variable. This option should only be deselected in exceptional cases, for example, when the target controller has sufficient and fast RAM, and speed is of the highest importance. This option can only be set on project level.

Const guard expression FPT

Defines the guard expression function pointer table as a `const` variable. This option should only be deselected in exceptional cases, for example, when the target controller has sufficient and fast RAM, and speed is of the highest importance. This option can only be set on project level.

Const action expression FPT

Defines the action expression function pointer table as a `const` variable. This option should only be deselected in exceptional cases, for example, when the target controller has sufficient and fast RAM, and speed is of the highest importance. This option can only be set on project level.

Generate digital signature

Includes a digital signature in the generated code. This option should normally be deselected, because it produces different files even if the model is unchanged. This option can only be set on project level.

Event parameter mechanism

Select the signature of `VSDeduct`. This option can only be set on project level.

Choose between:

Variable argument list

Uses the default `...` syntax.

Project-related union

Creates a union on project level to hold the event parameters.

System-related union

Creates a union on system level to hold the event parameters.

Insert type casts in functional expressions

Adds a type cast at the right-hand side of expressions, to reduce warnings in the generated code. This option can only be set on project level.

Insert void statements for unused formal parameters

Adds a type cast in action calls, to reduce warnings in the generated code. This option can only be set on project level.

Generated identifier prefix

Adds a prefix to the types (etc) of generated identifiers. This option can only be set on project level.

Generate C++ code

By default, the Hierarchical Coder generates C API code. Selecting this option makes the Coder instead generate C++ API code. This option can only be set on project level.

Project namespace

Specify the name of the namespace used for common project types, etc, when generating C++ code. This option can only be set on project level.

System namespace

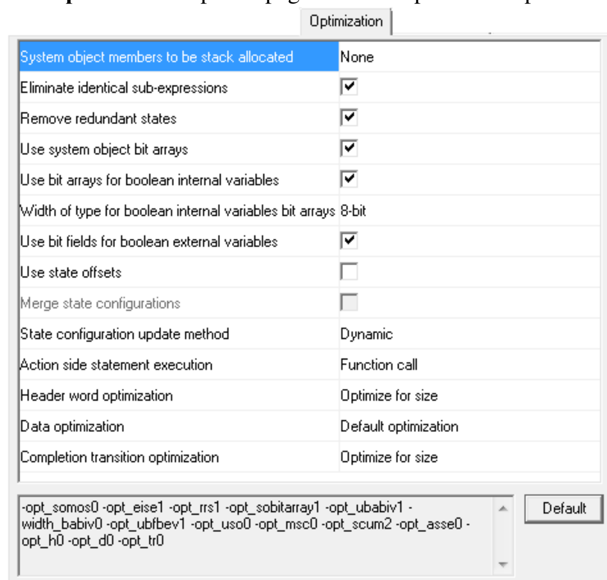
Specify the name of the namespace used for the system class. This option can only be set on system level.

Default

Restores the options to their default settings.

Hierarchical Coder Options dialog box : Optimization

The **Optimization** options page contains options for optimization.



Use this page to optimize the code generated by the Hierarchical Coder. The display area under the options shows the resulting command line for the code generation.

This options page can only be accessed on project level.

System object members to be stack allocated

Select which members of system objects to allocate on the stack.

Choose between:

None

No members of system objects are allocated on the stack.

Uninitialized candidates

Uninitialized members of system objects are allocated on the stack.

All candidates

All members of system objects that can be allocated on the stack are allocated on the stack.

Eliminate identical sub-expressions

Eliminates identical subexpressions in generated code guard and action expressions.

Remove redundant states

Eliminates all states that can be removed without changing the behavior of the model.

Use system object arrays

Makes the Hierarchical Coder use bit arrays for system objects. This makes the code size smaller, but has a negative effect on speed.

Use bit arrays for boolean internal variables

Makes the Hierarchical Coder use bit arrays for internal Boolean variables. This makes the code size smaller, but has a negative effect on speed.

Width of type for boolean internal variables bit arrays

Specify the width of the bit array type.

Choose between:

8-bit

The bit array type is 8-bit.

16-bit

The bit array type is 16-bit.

32-bit

The bit array type is 32-bit.

64-bit

The bit array type is 64-bit.

Use bitfields for boolean external variables

Makes the Hierarchical Coder use bitfields for external Boolean variables. This makes the code size smaller, but has a negative effect on speed.

Use state offsets

Makes the Hierarchical Coder use state offsets instead of fixed numbers. This makes the code size smaller, but has a negative effect on speed. This option can only be selected if the option **Use bitfields for boolean external variables** has been selected.

Merge state configurations

Makes the Hierarchical Coder merge state configurations. This makes the code size smaller, but has a negative effect on speed.

State configuration update method

Select which state configuration update method to use.

Choose between:

All

Always updates the entire state configuration. This makes your application smaller.

Partly Dynamic

Dynamically calculates and updates some parts of the state configuration depending on fired trans reactions. This might make your application faster but larger.

Dynamic

Dynamically calculates and updates the state configuration depending on fired trans reactions. This makes your application faster but larger.

Action side statement execution

Determine how the Hierarchical Coder executes action side statements.

Choose between:

Function call

Action side statements are implemented as functions. This makes your application smaller.

Inline

Action side statements are inlined. This makes your application faster but larger.

Header word optimization

Select the method for optimizing header word extraction.

Choose between:

Optimize for size

Optimizes header word extraction in a way that makes your application smaller.

Optimize for speed

Optimizes header word extraction in a way that makes your application faster.

Default optimization

The default method optimizes header word extraction in a way that makes your application mostly smaller but a little slower.

Data optimization

Select the method for optimizing data header extraction.

Choose between:

Default optimization

The default method optimizes data header extraction in a way that makes your application smaller.

Optimize for speed

Optimizes data header extraction in a way that makes your application faster.

Completion transition optimization

Select the method for optimizing completion transition handling.

Choose between:

Optimize for size

Optimizes completion transition handling in a way that makes your application smaller.

Optimize for speed

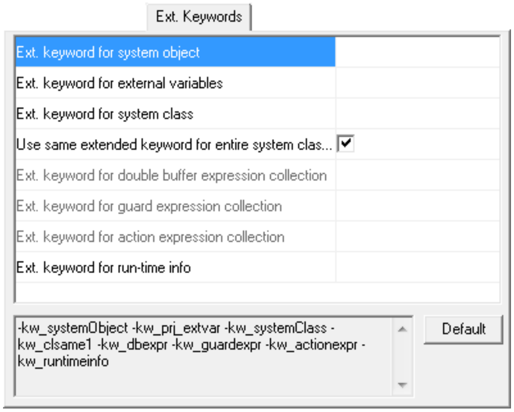
Optimizes completion transition handling in a way that makes your application faster.

Default

Restores the options to their default settings.

Hierarchical Coder Options dialog box : Extended Keywords

The **Extended Keywords** options page contains options for extended keywords.



Use this page to make extended keywords settings for the Hierarchical Coder. The display area under the options shows the resulting command line for the code generation.

This options page can be accessed on both project level and system level. Exactly which options you can set depends on the level you are setting options on.

Extended keyword for system object

Specify an extended keyword string for the system object variables (variable data). This option can only be set on project level.

Extended keyword for external variables

Specify an extended keyword string for external variables (variable data). This option can be set on both project level and system level.

Extended keyword for system class

Specify an extended keyword string for the system class variables (variable data). This option can only be set on project level.

Extended keyword for entire system class model

Makes the Hierarchical Coder use same extended keyword for double buffer, guard and action expression collections. If this option is selected, these do not need to be set individually. This option can only be set on project level.

Extended keyword for double buffer variable

Specify an extended keyword for variable buffering data (constant data). This option can only be set on project level.

Extended keyword for guard expression collection

Specify an extended keyword guard expression data (constant data). This option can only be set on project level.

Extended keyword for action expression collection

Specify an extended keyword for the action expression collection variables (constant data). This option can only be set on project level.

Extended keyword for runtime information

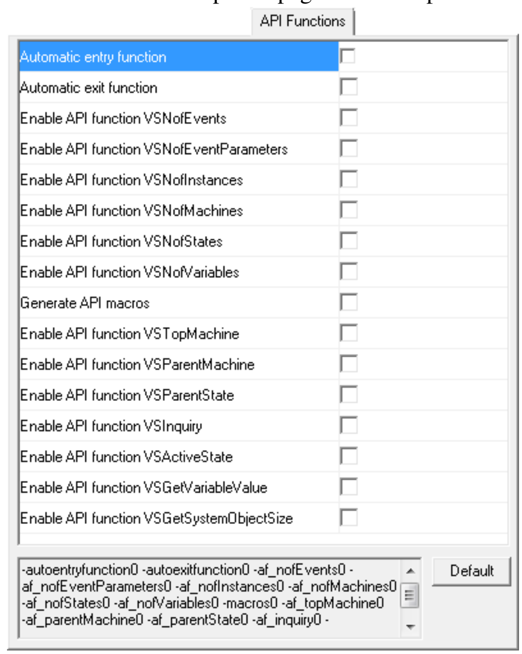
Specify an extended keyword string for the runtime information struct variable (constant data). By default, the runtime information struct only contains the digital signature for the project. This option can only be set on project level.

Default

Restores the options to their default settings.

Hierarchical Coder Options dialog box : API Functions

The **API Functions** options page contains options for API functions.



Use this page to make API function settings for the Hierarchical Coder and to enable specific API functions. The display area under the options shows the resulting command line for the code generation.

This options page can only be accessed on project level.

Automatic entry function

Adds a function call to a predefined function whenever a state is entered. This can help you debug the state machine.

Automatic exit function

Adds a function call to a predefined function whenever a state is exited. This can help you debug the state machine.

Generate API macros

Makes the Hierarchical Coder generate a set of API macros that might be useful for conditional compilation.

Enable API function *function*

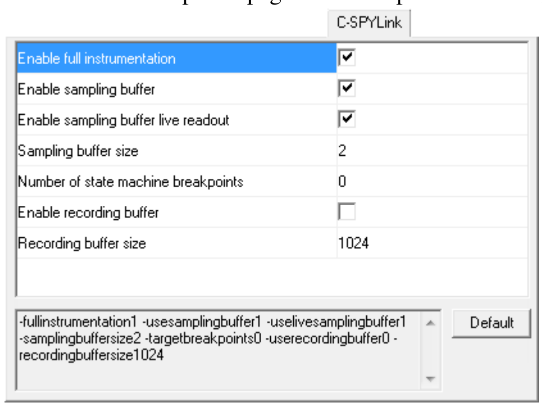
Enables a specific HCoder API function. See *Descriptions of the HCoder API functions*, page 474.

Default

Restores the options to their default settings.

Hierarchical Coder Options dialog box : C-SPYLink

The **C-SPYLink** options page contains options for debugging using C-SPYLink.



Use this page to make C-SPYLink settings for the Hierarchical Coder. The display area under the options shows the resulting command line for the code generation.

This options page can be accessed on both project level and system level. Depending on which level you set options on, different sets of options are available.

See also *Debugging design models using C-SPYLink*, page 759.

Enable using shared DLIB breakpoint

Makes the generated code use the shared breakpoint available in the DLIB runtime environment. If the number of breakpoints is limited, this helps to preserve the number of allocated breakpoints. Do not use this option with the legacy CLIB runtime environment. This option can only be set on project level.

Enable using ARM EABI shared semi-hosting breakpoint

Makes the generated code use the shared semi-hosting breakpoint available in the Arm EABI-specific runtime environment. If the number of breakpoints is limited, this helps to preserve the number of allocated breakpoints. This option requires IAR Embedded Workbench® for Arm 5.10 or later. This option can only be set on project level.

Suppress C-SPYLink common files

Prevents multiple C-SPYLink files from being generated when you are using two or more projects in the same linked image together with C-SPYLink. This option can only be set on project level.

Enable full instrumentation

Extracts a maximum amount of debug information from your model. This option causes a small increase in code size and a significant reduction in execution speed. This option can only be set on system level.

Enable sampling buffer

Enables on-target sampling buffers for a single macrostep. C-SPYLink will be able to extract large amounts of debug information from your model. This option causes an increase in code size and a small reduction in execution speed. If sequence recording is used, the speed reduction will be larger. Use the option **Sampling buffer size** to set the size of the buffer.

This option can only be set on system level.

Enable sampling buffer readout

Reads data from the sampling buffer while the target application is executing. The target controller must support live read. This option can only be set on system level.

Sampling buffer size

Set the number of elements in the sampling buffer for C-SPYLink. If the value is too low, you can only see the event that triggered the most recent transition and the states after that microstep. If the value is too high, the target application might run out of memory. This option does not change the behavior of the model.

This option can only be set on system level.

Number of state machine breakpoints

Set the number of available breakpoints for C-SPYLink on the target controller. Using this option consumes memory. This option does not change the behavior of the model.

This option can only be set on system level.

Enable recording buffer

Makes it possible to make recordings (execution logs) at almost full speed. This option also makes it possible to display sampled data back. Use the option **Recording buffer size** to set the size of the buffer.

This option can only be set on system level.

Recording buffer size

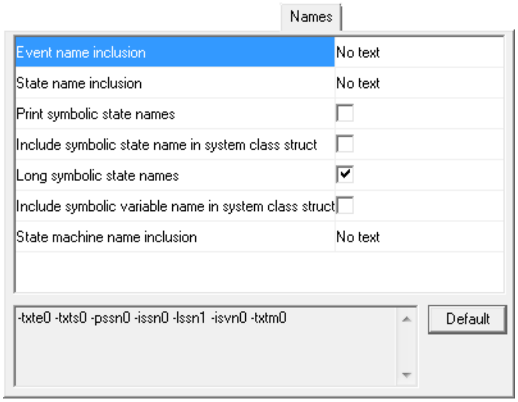
Set the number of elements in the recording buffer for C-SPYLink. This option can only be set on system level.

Default

Restores the options to their default settings.

Hierarchical Coder Options dialog box : Names

The **Names** options page contains options for including text associated with states, events, and actions in the generated code.



Use this page to make name settings for the Hierarchical Coder. The display area under the options shows the resulting command line for the code generation.

This options page can be accessed on both project level and system level. Exactly which options you can set depends on the level you are setting options on.

Event name inclusion

Specify the amount of text associated with events to include in the generated code. This option can only be set on system level.

Choose between:

No text

Includes no text associated with events in the generated code.

Names included

Includes the names of the events in the generated code.

Explanations included

Includes the descriptions of the events in the generated code.

Names and explanations

Includes both the names and the descriptions of the events in the generated code.

State name inclusion

Specify the amount of text associated with states to include in the generated code. This option can only be set on system level.

Choose between:

No text

Includes no text associated with states in the generated code.

Names included

Includes the names of the states in the generated code.

Explanations included

Includes the descriptions of the states in the generated code.

Names and explanations

Includes both the names and the descriptions of the states in the generated code.

Print symbolic state names

Makes the Hierarchical Coder generate symbolic state names. This option can only be set on system level.

Include symbolic state name in system class struct

Makes the Hierarchical Coder include the symbolic state names in the system class struct. This option can only be set on system level.

Long symbolic state names

Makes the Hierarchical Coder generate long names for states. Deselect this option if you want the Coder to generate short names. This option can only be set on system level.

Include symbolic variable name in system class struct

Makes the Hierarchical Coder include the symbolic variable names in the system class struct. This option can only be set on system level.

State machine name inclusion

Specify the amount of text associated with state machines to include in the generated code. This option can only be set on system level.

Choose between:

No text

Includes no text associated with state machines in the generated code.

Names included

Includes the names of the state machines in the generated code.

Explanations included

Includes the descriptions of the state machines in the generated code.

Names and explanations

Includes both the names and the descriptions of the state machines in the generated code.

Default

Restores the options to their default settings.

Type identifiers

The type identifiers are defined in the Hierarchical Coder-generated file `project.h`.

These are the available type identifiers:

Type identifiers	Description
`VSTriggerType`	Event data type
`VSDBExptrType`	Variable buffering expression type
`VSGuardExprType`	Guard expression data type
`VSStateType`	State data type
`VSDestinationStateType`	State data type.
`VSActionExprType`	Action expression data type.
`VSSignalQueueType`	Signal queue data type.

Table 27: Type identifiers — HCoder

Type identifiers	Description
`VSInstanceType`	Instance data type.
`VSVariableType`	Variable data type.
`VSEventParameterType`	Event parameter data type.
`VSMachineType`	State machine data type.

Table 27: Type identifiers — HCoder (Continued)

Transition rule data format

The transition rule data format is used for storing transitions in the local code layer. Each transition rule consists of one rule data header word and one rule data element for each element of the transition rule. For more information about the transition rule data format, see *Transition rule data format*, page 699.

Hierarchical Coder command line options

- Introduction to invoking the HCoder using command line options
- Summary of Hierarchical Coder options
- Descriptions of Hierarchical Coder options.

Introduction to invoking the HCoder using command line options

Learn more about:

- *Briefly about invoking the Hierarchical Coder*, page 517
- *Invocation syntax for the Hierarchical Coder*, page 518

BRIEFLY ABOUT INVOKING THE HIERARCHICAL CODER

You can set Hierarchical Coder options either in the Navigator—using the **Hierarchical Coder Options** dialog box—or via the command line using command line options.

A Coder option is either a project option or a system option. In general, project options affect the project and all systems part of it. System options only affect the systems for which they are specified.

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 apply to all systems.

Coder options are categorized based on these types:

Enumeration options	[E]	Require an argument.
Integral options	[I]	Require an argument.
Text options	[T]	Supplying an argument is optional.
Boolean options	[B]	Supplying an argument is optional. If no argument is supplied, the option will be set to its default value.

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.

For a complete list of available Hierarchical Coder options, run the `HCoder.exe` from the command prompt.

INVOCATION SYNTAX FOR THE HIERARCHICAL CODER

This is the invocation syntax for starting the Hierarchical Coder from the command line:

`HCoder.exe` *`vsp_file`* `[--l] [--@`*`option-file`*`] -`*`option`*`[`*`argument`*`]*`

Where:

`--l` loads options from the `vtg` file associated with the specified `vsp` file.

`--@`*`option-file`* loads 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 `//`.

Example 1

```
HCoder.exe Mobile.vsp
```

Description:	Generates code for the project and stores it in the file `Mobile.vsp`.

Example 2

```
HCoder.exe Mobile.vsp -Vmobile1 -txte3 -txts3 -Vmobile2
```

Description:	Generates code for the project, which contains the systems `Mobile1` and `Mobile2`. In addition, the system `Mobile1` will be generated with names and descriptions for events, states, and action functions.

Example 3

```
HCoder.exe Mobile.vsp --@MobileSetup.txt -Vmobile -txte3 -txts3
```

Description:	Generates code for the project, which contains the system `Mobile`. In addition, the system `Mobile` will be generated with names and descriptions for events, states, and action functions.

Summary of Hierarchical Coder options

This table summarizes the Hierarchical Coder command line options:

Command line option	Description
-af_activeState	Enables the HCoder API function VSActiveState. [Project option]
-af_gsos	Enables the HCoder API function VSGetSystemObjectSize. [Project option]
-af_gvv	Enables the HCoder API function VSGetVariableValue. [Project option]
-af_inquiry	Enables the HCoder API function VSInquiry. [Project option]
-af_nofEventParameters	Enables the HCoder API function VSNofEventParameters. [Project option]
-af_nofEvents	Enables the HCoder API function VSNofEvents. [Project option]
-af_nofInstances	Enables the HCoder API function VSNofInstances. [Project option]
-af_nofMachines	Enables the HCoder API function VSNofMachines. [Project option]
-af_nofStates	Enables the HCoder API function VSNofStates. [Project option]
-af_nofVariables	Enables the HCoder API function VSNofVariables. [Project option]
-af_parentMachine	Enables the HCoder API function VSParentMachine. [Project option]
-af_parentState	Enables the HCoder API function VSParentState. [Project option]
-af_topMachine	Enables the HCoder API function VSTopMachine. [Project option]
-armsemihostingbreakpoint	Determines whether the generated code uses the shared Arm EABI semi-hosting breakpoint. [Project option]
-autoentryfunction	Adds a call to a predefined function whenever a state is entered. [Project option]
-autoexitfunction	Adds a call to a predefined function whenever a state is exited. [Project option]

Table 28: Hierarchical Coder command line options

Command line option	Description
`-constactionfpt`	Determines whether the action expression function pointer table is defined as a `const` variable. [Project option]
`-constguardfpt`	Determines whether the guard expression function pointer table is defined as a `const` variable. [Project option]
`-constsc`	Determines whether system class variables are defined as `const` variables. [Project option]
`-constvbfpt`	Determines whether the variable buffer expression function pointer table is defined as a `const` variable. [Project option]
`-cspylink`	Determines whether the generated code can be debugged using C-SPYLink. [Project option]
`-D`	Specifies the data width for data types for the entire project. [Project option]
`-dlibbreakpoint`	Determines whether the generated code uses the shared DLIB breakpoint. [Project option]
`-dso`	Allocates systems objects dynamically instead of statically. [Project option]
`-epm`	Selects the signature of `VSDeduct`. [Project option]
`-exclude`	Excludes a system from build.
`-fullinstrumentation`	Controls the amount of debug information that C-SPYLink can extract from your model. [System option]
`-funcexph`	Specifies how the Hierarchical Coder should handle functional expressions. [Project option]
`-gds`	Determines whether the Hierarchical Coder includes a digital signature in the generated code. [Project option]
`-gip`	Adds a prefix to the types (etc) of generated identifiers. [Project option]
`-H`	Specifies the name of the header file that contains system-level model declarations. [System option]
`-ipev`	Specifies how to initialize external variables. [Project option]
`-isev`	Specifies how to initialize system-external variables. [Project option]
`-issn`	Includes the symbolic state names in the system class struct. [System option]
`-isvn`	Includes the symbolic variable names in the system class struct. [System option]

Table 28: Hierarchical Coder command line options (Continued)

Command line option	Description
`-itcfe`	Adds a type cast at the right-hand side of functional expressions. [Project option]
`-ivsufp`	Inserts void statements for unused formal parameters. [Project option]
`-kw_actionexpr`	Specifies an extended keyword string for the action expression collection variables. [Project option]
`-kw_clsame`	Uses the same extended keyword for the entire system class model. [Project option]
`-kw_dbexpr`	Specifies an extended keyword for variable buffering data. [Project option]
`-kw_guardexpr`	Specifies an extended keyword string for the guard expression collection variables. [Project option]
`-kw_prj_extvar`	Specifies an extended keyword string for external variables in the entire project. [Project option]
`-kw_runtimeinfo`	Specifies an extended keyword string for the runtime information `struct` variable. [Project option]
`-kw_sys_extvar`	Specifies an extended keyword string for external variables. [System option]
`-kw_systemClass`	Specifies an extended keyword string for the system class variables. [Project option]
`-kw_systemObject`	Specifies an extended keyword string for the system object variables. [Project option]
`-lssn`	Generate long symbolic names for states. [System option]
`-macros`	Generates HCoder API macros. [Project option]
`-no_warnings`	Determines whether warnings should be disabled. [Project option]
`-opt_asse`	Determines how to execute action side statements. [Project option]
`-opt_d`	Determines how to optimize data header extraction. [Project option]
`-opt_eise`	Eliminates identical sub-expressions. [Project option]
`-opt_h`	Determines how to optimize header word optimization. [Project option]
`-opt_msc`	Merges state configurations. [Project option]
`-opt_rrs`	Removes redundant states. [Project option]

Table 28: Hierarchical Coder command line options (Continued)

Command line option	Description
-opt_scum	Determines how to update state configurations. [Project option]
-opt_sobitarray	Uses bit arrays for system objects. [Project option]
-opt_somos	Allocates system object members on the stack. [Project option]
-opt_tr	Determines how to optimize completion transitions. [Project option]
-opt_ubabiv	Use bit arrays for Boolean internal variables. [Project option]
-opt_ubfbev	Use bitfields for Boolean external variables. [Project option]
-opt_uso	Uses state offsets instead of fixed numbers. [Project option]
-path	Specifies the output path for all generated project files. [Project option]
-projectheader	Specifies the name of the header file that contains project-level model declarations. [Project option]
-projectsource	Specifies the name of the source file that contains project-level model definitions. [Project option]
-pssf	Merges all project and system source files into the main project source file. [Project option]
-pssn	Generates symbolic state names. [System option]
-R	Specifies a name for a report file to contain information about the project. [Project option]
-recordingbuffersize	Specifies the number of elements in the recording buffer for C-SPYLink. [System option]
-riins	Determines whether the internal instances can be reinitialized or not. [Project option]
-S	Specifies the name of the source file that contains system-level model definitions. [System option]
-samplingbuffersize	Specifies the number of elements in the sampling buffer for C-SPYLink. [System option]
-ssewi	Sends the start event automatically after VSInitAll has been executed. [Project option]
-siss	Initializes static storage with zeros. [Project option]
-spath	Specifies the output path for all generated system files. [System option]

Table 28: Hierarchical Coder command line options (Continued)

Command line option	Description
-suppress_cspylink_common_files	Controls how multiple C-SPYLink files are generated when you are using two or more projects in the same linked image. [Project option]
-targetbreakpoints	Specifies the number of available breakpoints for C-SPYLink on the target controller. [System option]
-txte	Controls the amount of text associated with events to include in the generated code. [System option]
-txtm	Controls the amount of text associated with state machines to include in the generated code. [System option]
-txts	Controls the amount of text associated with states to include in the generated code. [System option]
-uselivesamplingbuffer	Determines whether C-SPYLink can read data from the sampling buffer while the target application is executing. [System option]
-usepop	Determines whether the Hierarchical Coder uses the same output path for system files as the path specified for all project files. [System option]
-userecordingbuffer	Determines whether to use a recording buffer. [System option]
-usesamplingbuffer	Controls on-target sampling buffers for a single macrostep. [System option]
-V	Specifies the system that the following system options apply to. [System option]
-variant	Specifies which variant to generate code for. [Project option]
-warnings_affect_exit_code	Determines whether warnings generate a non-zero exit code. [Project option]
-warnings_are_errors	Determines whether all warnings are reclassified as errors. [Project option]
-width_babiv	Sets the width of the bit array type. [Project option]

Table 28: Hierarchical Coder command line options (Continued)

Descriptions of Hierarchical Coder options

The following pages give detailed reference information about each Hierarchical Coder command line option.

-af_activeState

Syntax

`-af_activeState{0|1}`

Parameters

`0` (default)	The API function is not enabled.
`1`	The API function is enabled.

Scope

Project level.

Description

Specifies whether to enable the API function `VSActiveState`. The function returns the active state of a specific machine. If the machine is not active, `VS_StateUndefined` is returned.

See also

VSActiveState, page 474.

Project>Options>Code Generation>*project*>API Functions>Enable API function VSActiveState

-af_gsos

Syntax

`-af_gsos{0|1}`

Parameters

`0` (default)	The API function is not enabled.
`1`	The API function is enabled.

Scope

Project level.

Description

Specifies whether to enable the API function `VSGetSystemObjectSize`. The function returns the size in bytes of system objects of specified system class.

See also

VSGetSystemObjectSize, page 479.

Project>Options>Code Generation>*project*>API Functions>Enable API function VSGetSystemObjectSize

-af_gvv

Syntax `-af_gvv{0|1}`

Parameters

`0` (default)	The API function is not enabled.
`1`	The API function is enabled.

Scope Project level.

Description Specifies whether to enable the API function `VSVariableValue`. The function prints the value of a specified variable to a string.

See also *VSVariableValue*, page 490.

Project>Options>Code Generation>*project*>API Functions>Enable API function VSVariableValue

-af_inquiry

Syntax `-af_inquiry{0|1}`

Parameters

`0` (default)	The API function is not enabled.
`1`	The API function is enabled.

Scope Project level.

Description Specifies whether to enable the API function `VSInquiry`. The function tests whether an event is active with regards to state conditions.

See also *VSInquiry*, page 480.

Project>Options>Code Generation>*project*>API Functions>Enable API function VSInquiry

-af_nofEventParameters

Syntax `-af_nofEventParameters{0|1}`

Parameters

`0` (default)	The API function is not enabled.
`1`	The API function is enabled.

Scope Project level.

Description Specifies whether to enable the API function `VSNofEventParameters`. The function returns the number of event parameters for a specified event within a system.

See also *VSNofEventParameters*, page 483.

Project>Options>Code Generation>*project*>API Functions>Enable API function VSNofEventParameters

-af_nofEvents

Syntax `-af_nofEvents{0|1}`

Parameters

`0` (default)	The API function is not enabled.
`1`	The API function is enabled.

Scope Project level.

Description Specifies whether to enable the API function `VSNofEvents`. The function returns the number of defined events for a system.

See also *VSNofEvents*, page 484.

Project>Options>Code Generation>*project*>API Functions>Enable API function VSNofEvents

-af_nofInstances

Syntax `-af_nofInstances{0|1}`

Parameters

`0` (default)	The API function is not enabled.
`1`	The API function is enabled.

Scope Project level.

Description Specifies whether to enable the API function `VSNofInstances`. The function returns the number of internal instances in a system.

See also *VSNofInstances*, page 484.

Project>Options>Code Generation>*project*>API Functions>Enable API function VSNofInstances

-af_nofMachines

Syntax `-af_nofMachines{0|1}`

Parameters

`0` (default)	The API function is not enabled.
`1`	The API function is enabled.

Scope Project level.

Description Specifies whether to enable the API function `VSNofMachines`. The function returns the number of defined machines for a system.

See also *VSNofMachines*, page 484.

Project>Options>Code Generation>*project*>API Functions>Enable API function VSNofMachines

-af_nofStates

Syntax

`-af_nofStates{0|1}`

Parameters

`0` (default)	The API function is not enabled.
`1`	The API function is enabled.

Scope

Project level.

Description

Specifies whether to enable the API function `VSNofStates`. The function returns the number of defined states for a system.

See also

VSNofStates, page 485.

Project>Options>Code Generation>*project*>API Functions>Enable API function VSNofStates

-af_nofVariables

Syntax

`-af_nofVariables{0|1}`

Parameters

`0` (default)	The API function is not enabled.
`1`	The API function is enabled.

Scope

Project level.

Description

Specifies whether to enable the API function `VSNofVariables`. The function returns the number of defined variables in scope for a system.

See also

VSNofVariables, page 485.

Project>Options>Code Generation>*project*>API Functions>Enable API function VSNofVariables

-af_parentMachine

Syntax `-af_parentMachine{0|1}`

Parameters

`0` (default)	The API function is not enabled.
`1`	The API function is enabled.

Scope Project level.

Description Specifies whether to enable the API function `VSParentMachine`. The function returns the parent machine of the specified state.

See also *VSParentMachine*, page 486.

Project>Options>Code Generation>*project*>API Functions>Enable API function VSParentMachine

-af_parentState

Syntax `-af_parentState{0|1}`

Parameters

`0` (default)	The API function is not enabled.
`1`	The API function is enabled.

Scope Project level.

Description Specifies whether to enable the API function `VSParentState`. The function returns the parent state of the specified machine. For the top machine, `VS_StateUndefined` is returned.

See also *VSParentState*, page 486.

Project>Options>Code Generation>*project*>API Functions>Enable API function VSParentState

-af_topMachine

Syntax `-af_topMachine{0|1}`

Parameters

`0` (default)	The API function is not enabled.
`1`	The API function is enabled.

Scope Project level.

Description Specifies whether to enable the API function `VSTopMachine`. The function returns the top machine.

See also *VSTopMachine*, page 490.

Project>Options>Code Generation>*project*>API Functions>Enable API function VSTopMachine

-armsemihostingbreakpoint

Syntax `-armsemihostingbreakpoint{0|1}`

Parameters

`0` (default)	The generated code does not use the shared semi-hosting breakpoint available in the Arm EABI-specific runtime environment.
`1`	The generated code uses the shared semi-hosting breakpoint available in the Arm EABI-specific runtime environment.

Scope Project level.

Description Determines whether the generated code uses the shared semi-hosting breakpoint available in the Arm EABI-specific runtime environment. If the number of breakpoints is limited, using this breakpoint helps to preserve the number of allocated breakpoints. This option requires IAR Embedded Workbench® for Arm 5.10 or later.

See also -*dlibbreakpoint*, page 716.

Project>Options>Code Generation>*project*>C-SPYLink>Enable using ARM EABI shared semi-hosting breakpoint

-autoentryfunction

Syntax `-autoentryfunction{0|1}`

Parameters

`0` (default)	A function call to a predefined function is not added whenever a state is entered.
`1`	A function call to a predefined function is added whenever a state is entered.

Scope Project level.

Description Specifies whether to add a function call to a predefined function whenever a state is entered.

See also *VSProjectEnterState*, page 477.

Project>Options>Code Generation>*project*>API Functions>Automatic entry function

-autoexitfunction

Syntax `-autoexitfunction{0|1}`

Parameters

`0` (default)	A function call to a predefined function is not added whenever a state is exited.
`1`	A function call to a predefined function is added whenever a state is exited.

Scope Project level.

Description Specifies whether to add a function call to a predefined function whenever a state is exited.

See also *VSProjectLeaveState*, page 481.

Project>Options>Code Generation>*project*>API Functions>Automatic exit function

-constactionfpt

Syntax `-constactionfpt{0|1}`

Parameters

`0`	The action expression function pointer table is not defined as a `const` variable.
`1` (default)	Defines the action expression function pointer table as a `const` variable.

Scope Project level.

Description Determines whether the action expression function pointer table is defined as a `const` variable. This option should only be set to `0` in exceptional cases, for example, when the target controller has sufficient and fast RAM, and speed is of the highest importance.

See also -*constguardfpt*, page 532.

Project>Options>Code Generation>*project*>Code>Const action expression FPT

-constguardfpt

Syntax `-constguardfpt{0|1}`

Parameters

`0`	The guard expression function pointer table is not defined as a `const` variable.
`1` (default)	Defines the guard expression function pointer table as a `const` variable.

Scope Project level.

Description Determines whether the guard expression function pointer table is defined as a `const` variable. This option should only be set to `0` in exceptional cases, for example, when the target controller has sufficient and fast RAM, and speed is of the highest importance.

See also -*constactionfpt*, page 532.

Project>Options>Code Generation>*project*>Code>Const guard expression FPT

-constsc

Syntax `-constsc{0|1}`

Parameters

`0`	The system class is not defined as a `const` variable.
`1` (default)	Defines the system class as a `const` variable.

Scope Project level.

Description Determines whether the system class is defined as a `const` variable. This option should only be set to `0` in exceptional cases, for example, when the target controller has sufficient and fast RAM, and speed is of the highest importance.

See also -*constactionfpt*, page 532.

Project>Options>Code Generation>*project*>Code>Const system class

-constvbfpt

Syntax `-constvbfpt{0|1}`

Parameters

`0`	The variable buffer expression function pointer table is not defined as a `const` variable.
`1` (default)	Defines the variable buffer expression function pointer table as a `const` variable.

Scope Project level.

Description Determines whether the variable buffer expression function pointer table is defined as a `const` variable. This option should only be set to `0` in exceptional cases, for example, when the target controller has sufficient and fast RAM, and speed is of the highest importance.

See also -*constactionfpt*, page 532.

Project>Options>Code Generation>*project*>Code>Const guard variable buffer expression FPT

-cpp

Syntax `-cpp{0|1}`

Parameters

`0` (default)	Generates C code and API code.
`1`	Generates C++ code and API code.

Scope Project level.

Description Determines whether to generate C or C++ code and API code. If you specify `-cpp1`, you should also specify the namespace for all C++ code related to the system and the project namespace to use for C++ output.

See also -*namespace*, page 547 and -*projectnamespace*, page 555.

Project>Options>Code Generation>*project*>Code>Generate C++ code

-cppsourcefileext

Syntax `-cppsourcefileext`*`extension`*

Parameters

`extension`	The filename extension that IAR Visual State uses for generated C++ language source files.

Scope Project level.

Description Determines the filename extension that IAR Visual State uses for generated C++ language source files. By default, the filename extension is `cpp`.

Project>Options>Code Generation>*project*>File Output> C++ File extension

-cspylink

Syntax `-cspylink{0|1}`

Parameters

`0` (default)	Does not generate code to be debugged using C-SPYLink.

`1`	Generates code to be debugged using C-SPYLink.

Scope

Project level.

Description

Determines whether the generated code can be debugged using C-SPYLink.

See also

Debugging design models using C-SPYLink, page 759 and *-fullinstrumentation*, page 717.

Project>Options>Code Generation>*project*>Configuration>Generate for C-SPYLink

-D

Syntax

`-D{O|0|1|2}`

Parameters

`O` (default)	Uses the most optimal data widths for HCoder type definitions. The width is the smallest possible to reduce the use of variable and constant data.
`0`	Sets the data width of all HCoder types to 8 bits. If the target microcontroller handles 8-bit accesses well, this setting probably increases the execution speed.
`1`	Sets the data width of all HCoder types to 16 bits. If the target microcontroller handles 16-bit accesses well, this setting probably increases the execution speed.
`2`	Sets the data width of all HCoder types to 32 bits. If the target microcontroller handles 32-bit accesses well, this setting probably increases the execution speed.

Scope

Project level.

Description

Specifies the data width for internal data members. The default setting uses the smallest possible width. If you set the width to a smaller value than needed, the smallest possible value will be used.

See also

Type identifiers, page 515.

Project>Options>Code Generation>*project*>Code>Data width

-dlibbreakpoint

Syntax `-dlibbreakpoint{0|1}`

Parameters

`0` (default) The generated code does not use the shared breakpoint available in the DLIB runtime environment.

`1` The generated code uses the shared breakpoint available in the DLIB runtime environment.

Scope Project level.

Description Determines whether the generated code uses the shared breakpoint available in the DLIB runtime environment. If the number of breakpoints is limited, using this breakpoint helps to preserve the number of allocated breakpoints. Do not use this option with the legacy CLIB runtime environment.

See also *-armsemihostingbreakpoint*, page 709.

Project>Options>Code Generation>*project*>C-SPYLink>Enable using shared DLIB breakpoints

-dso

Syntax `-dso{0|1}`

Parameters

`0` (default) Allocates system objects statically.

`1` Allocates system objects dynamically.

Scope Project level.

Description Specifies whether to enable dynamic allocation of system objects.

Project>Options>Code Generation>*project*>Memory>Dynamic system objects

-epm

Syntax `-epm{0|1|2}`

Parameters

`0` (default)	Parameters are transferred to the API as a variable argument list. The API will copy the parameters to an internal buffer, using the macros `va_start`, `va_arg`, and `va_end`, defined in `stdarg.h`. Cannot be used in C++.
`1`	Parameters are transferred to the API as a pointer to a union. The API will copy the pointer to internal storage for later access. The contents of the union must remain valid throughout the entire macrostep. The type of the union is shared between all systems, which might cause the union to be larger than needed for the particular system.
`2`	Parameters are transferred to the API as a pointer to a union. The API will copy the pointer to internal storage for later access. The contents of the union must remain valid throughout the entire macrostep. A union type is generated for each system in order to minimize the size of the union. Default in C++.

Scope Project level.

Description Specifies how event parameters are transferred to the API.

Project>Options>Code Generation>*project*>Code>Event parameter mechanism

-exclude

Syntax `-exclude{0|1}`

Parameters

`0` (default)	Includes the system in code generation.
`1`	Excludes the system from code generation.

Scope System level.

Description Specifies whether to exclude the system from a build.

Project>Options>Code Generation>*system*>Configuration>Exclude System from build

-fullinstrumentation

Syntax

`-fullinstrumentation{0|1}`

Parameters

`0` (default)	Disables full instrumentation.
`1`	Enables full instrumentation, to extract a maximum amount of debug information.

Scope

System level.

Description

Controls the amount of debug information that C-SPYLink can extract from your model. Specifying `-fullinstrumentation1` causes a small increase in code size and a significant reduction in execution speed.

Project>Options>Code Generation>*system*>C-SPYLink>Enable full instrumentation

-funcexph

Syntax

`-funcexph{0|1|2}`

Parameters

`0` (default)	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.
`1`	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 alternative settings efficiently.
`2`	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 can convert the switch-case construct into a jump table, this might be the most efficient setting.

Scope Project level.

Description Specifies how the Hierarchical Coder should handle functional expressions (guard expressions and action expressions).

Project>Options>Code Generation>*project*>Code>Functional expression handling

-gds

Syntax `-gds{0|1}`

Parameters

`0` (default)	Does not include a digital signature in the generated code.
`1`	Includes a digital signature in the generated code.

Scope Project level.

Description Determines whether the Hierarchical Coder includes a digital signature in the generated code.

See also *Digital signatures for tracking inconsistencies*, page 74.

Project>Options>Code Generation>*project*>Code>Generate digital signature

-gip

Syntax `-gip`*`prefix`*

Parameters

`prefix`	A text string that will be used as a prefix.

Scope Project level.

Description Specifies a prefix to use for generated identifiers, except for identifiers for explicitly named elements such as external variables, constants, etc. This prefix will be used in both a lower-case and an upper-case version. Specifying different prefixes for different Visual State projects avoids name clashes when a compiler project contains code from several Visual State projects.

Project>Options>Code Generation>*project*>Code>Generated identifier prefix

-H

Syntax

`-H`*`file`*

Parameters

`file`	The name of the header file.

Scope

System level.

Description

Specifies the name of the header file that contains system-level model declarations. The name used by default is *`System`*`.h`.

Project>Options>Code Generation>*system*>File Output>System header file

-ipev

Syntax

`-ipev{0|1|2|3}`

Parameters

`0`	External variables are never initialized.
`1` (default)	Initializes external variables along with their definition.
`2`	Initializes external variables when system objects are initialized.
`3`	Initializes external variables when internal instances within system objects are initialized.

Scope

Project level.

Description

Specifies how to initialize project-external variables.

Project>Options>Code Generation>*project*>Code>Project external variable initialization

-isev

Syntax `-isev{0|1|2|3}`

Parameters

`0`	External variables are never initialized.
`1` (default)	Initializes external variables along with their definition.
`2`	Initializes external variables when system objects are initialized.
`3`	Initializes external variables when internal instances within system objects are initialized.

Scope Project level.

Description Specifies how to initialize system-external variables.

Project>Options>Code Generation>*project*>Code>System external variable initialization

-issn

Syntax `-issn{0|1}`

Parameters

`0` (default)	Does not include symbolic state names in the system class `struct`.
`1`	Includes symbolic state names in the system class `struct`.

Scope System level.

Description Specifies whether to include symbolic state names in the system class `struct`.

Project>Options>Code Generation>*system*>Names>Include symbolic state name in system class struct

-isvn

Syntax `-isvn{0|1}`

Parameters

`0` (default)	Does not include symbolic variable names in the system class `struct`.
`1`	Includes symbolic variable names in the system class `struct`.

Scope System level.

Description Specifies whether to include symbolic variable names in the system class `struct`.

Project>Options>Code Generation>*system*>Names>Include symbolic variable name in system class struct

-itcfe

Syntax `-itcfe{0|1}`

Parameters

`0` (default)	Does not insert typecasts in functional expressions.
`1`	Inserts typecasts in functional expressions.

Scope System level.

Description Specifies whether to insert typecasts in functional expressions. If `-itcfe1` is specified, a typecast is inserted on the right side of each assignment, and each actual parameter in an action function call is converted to the type of the corresponding formal parameter.

Specify `-itcfe1` to avoid warnings for typecasts in functional expressions when you compile generated code. However, be aware that this might hide logical errors in the design.

Project>Options>Code Generation>*project*>Code>Insert type casts in functional expressions

-ivsufp

Syntax `-ivsufp{0|1}`

Parameters

`0` (default)	Does not insert `void` statements for unused formal parameters.
`1`	Inserts `void` statements for unused formal parameters.

Scope Project level.

Description Specifies whether to insert `void` statements for unused formal parameters. Because of the overall design of the generated code, some functions might include formal parameters that are not used in the function body. To avoid compiler warnings, specify `-ivsufp1` to prepend the body with `void` statements of the form `(void) x;`, where `x` is the unused formal parameter.

Project>Options>Code Generation>*project*>Code>Insert void statements for unused formal parameters

-kw_actionexpr

Syntax `-kw_actionexpr`*`keyword`*

Parameters

`keyword`	A string that will be used as a keyword.

Scope Project level.

Description Specifies an extended keyword string for the action expression collection variables (constant data).

Project>Options>Code Generation>*project*>Ext. Keywords>Extended keyword for action expression collection

-kw_clsame

Syntax `-kw_clsame{0|1}`

Parameters

`0`	Uses different extended keywords for all system class model members.

`1` (default)	Uses the same extended keyword for all system class model members.

Scope

Project level.

Description

Specifies whether to use the same extended keyword for all system class model members. If you specify `-kw_clsame1`, double buffer expression, guard expression, and action expression collections will use the same keyword as for the system class `struct`.

Project>Options>Code Generation>*project*>Ext. Keywords>Use same extended keyword for entire system class model

-kw_dbexpr

Syntax

`-kw_dbexpr`*`keyword`*

Parameters

`keyword`	A string that will be used as a keyword.

Scope

Project level.

Description

Specifies an extended keyword string for the double buffer expression collection variables (constant data).

Project>Options>Code Generation>*project*>Ext. Keywords>Extended keyword for double buffer expression collection

-kw_guardexpr

Syntax

`-kw_guardexpr`*`keyword`*

Parameters

`keyword`	A string that will be used as a keyword.

Scope

Project level.

Description

Specifies an extended keyword string for the guard expression collection variables (constant data).

Project>Options>Code Generation>*project*>Ext. Keywords>Extended keyword for guard expression collection

-kw_prj_extvar

Syntax `-kw_prj_extvar`*`keyword`*

Parameters

`keyword`	A string that will be used as a keyword.

Scope Project level.

Description Specifies an extended keyword string for external variables (variable data) in the entire project.

See also -*kw_sys_extvar*, page 546.

Project>Options>Code Generation>*project*>Ext. Keywords>Extended keyword for external variables

-kw_runtimeinfo

Syntax `-kw_runtimeinfo`*`keyword`*

Parameters

`keyword`	A string that will be used as a keyword.

Scope Project level.

Description Specifies an extended keyword string for the runtime information `struct` variable (constant data). By default, the runtime information `struct` only contains the digital signature for the project.

Project>Options>Code Generation>*project*>Ext. Keywords>Extended keyword for runtime info

-kw_sys_extvar

Syntax `-kw_sys_extvar`*`keyword`*

Parameters

`keyword`	A string that will be used as a keyword.

Scope System level.

Description Specifies an extended keyword string for external variables (variable data) in a system.

See also *-kw_prj_extvar*, page 545.

Project>Options>Code Generation>*system*>Ext. Keywords>Extended keyword for external variables

-kw_systemClass

Syntax `-kw_systemClass`*`keyword`*

Parameters

`keyword`	A string that will be used as a keyword.

Scope Project level.

Description Specifies an extended keyword string for the system class variables (constant data).

Project>Options>Code Generation>*project*>Ext. Keywords>Extended keyword for system class

-kw_systemObject

Syntax `-kw_systemObject`*`keyword`*

Parameters

`keyword`	A string that will be used as a keyword.

Scope Project level.

Description Specifies an extended keyword string for the system object variables (variable data).

Project>Options>Code Generation>*project*>Ext. Keywords>Extended keyword for system object

-lssn

Syntax `-lssn{0|1}`

Parameters

`0`	Generates short names for states.
`1` (default)	Generates long names for states.

Scope System level.

Description Specifies whether to generate long symbolic names for states.

Project>Options>Code Generation>*project*>Names>Long symbolic state names

-macros

Syntax `-macros{0|1}`

Parameters

`0` (default)	API macros are not generated.
`1`	API macros are generated.

Scope Project level.

Description Specifies whether to generate a set of API macros.

Project>Options>Code Generation>*project*>API Functions >Generate API macros

-namespace

Syntax `-namespace`*`name`*

Parameters

`name`	The name of the system namespace used by generated C++ code.

Scope System level.

Description Specifies the C++ namespace for all code related to the system. By default, the namespace is "".

Project>Options>Code Generation>*system*>Code Generation>System namespace

-no_warnings

Syntax `-no_warnings{0|1}`

Parameters

`0` (default)	Warnings are issued.
`1`	Warnings are disabled and cannot affect the exit code.

Scope Project level.

Description Determines whether warnings should be disabled.

See also *-warnings_are_errors*, page 566

Project>Options>Code Generation>*project*>Configuration>Ignore warnings

-opt_asse

Syntax `-opt_asse{0|1}`

Parameters

`0` (default)	Action side statements are implemented as functions. This minimizes the application size.
`1`	Action side statements are inlined. This maximizes the application's execution speed.

Scope Project level.

Description Specifies how statements that belong to the action side of a transition/reaction are handled.

Project>Options>Code Generation>*project*>Optimization>Action side statement execution

-opt_d

Syntax `-opt_d{0|1}`

Parameters

`0` (default)	Optimizes data header extraction to make the application smaller.
`1`	Optimizes data header extraction to make the application faster.

Scope Project level.

Description Specifies how to optimize data header extraction.

Project>Options>Code Generation>*project*>Optimization>Data optimization

-opt_eise

Syntax `-opt_eise{0|1}`

Parameters

`0`	Identical subexpressions are not eliminated.
`1` (default)	Eliminates identical subexpressions. This might make the application smaller but slightly slower.

Scope Project level.

Description Specifies whether to eliminate identical subexpressions in compound guard and action expressions.

Project>Options>Code Generation>*project*>Optimization>Eliminate identical sub-expressions

-opt_h

Syntax `-opt_h{0|1|2}`

Parameters

`0`	Optimizes header word extraction to make the application smaller.
`1` (default)	Optimizes header word extraction to make the application mostly smaller but a little slower.
`2`	Optimizes header word extraction to make the application faster.

Scope Project level.

Description Specifies how to optimize header word extraction.

Project>Options>Code Generation>*project*>Optimization>Header word optimization

-opt_msc

Syntax `-opt_msc{0|1}`

Parameters

`0` (default)	State configurations are not merged.
`1`	Merges states from different internal configurations into one configuration. This minimizes RAM usage.

Scope Project level.

Description Specifies whether to merge state configurations. If you specify `-opt_msc1`, no machine can have more than 14 child states, otherwise code generation stops. If the project contains shallow history states or deep history states, the limit is 13, and if the project contains both kinds of history states, the limit is 12.

Project>Options>Code Generation>*project*>Optimization>Merge state configurations

-opt_rrs

Syntax

`-opt_rrs{0|1}`

Parameters

`0`	Redundant states are not removed.
`1` (default)	Removes redundant states.

Scope

Project level.

Description

Specifies whether to remove redundant states. A redundant state is a state that:

- has no sibling states or pseudo-states (except for an optional initial state with an empty default reaction)
- has no entry
- has no internal or exit reactions
- is not used as direct source, main source or destination in any transition

If redundant states are removed, all synchronizations to the state are changed to synchronizations to the parent state.

Project>Options>Code Generation>*project*>Optimization>Remove redundant states

-opt_scum

Syntax

`-opt_scum{0|1|2}`

Parameters

`0`	Always updates the entire state configuration, to make the application smaller.
`1`	Dynamically excludes some part of the state configuration from updating that was not affected by firing of transitions or reactions. This might make the application faster but slightly larger.
`2` (default)	Dynamically calculates and updates the part of the state configuration that was actually affected by firing of transitions or reactions. This might increase speed but also increase the size of code and variable data.

Scope

Project level.

Description

Determines how to update state configurations.

Project>Options>Code Generation>*project*>Optimization>State configuration update method

-opt_sobitarray

Syntax

`-opt_sobitarray{0|1}`

Parameters

`0`	Bit arrays are not used. This increases application speed.
`1` (default)	Uses bit arrays. This minimizes the size of variable data.

Scope

Project level.

Description

Specifies whether to use bit arrays for system object members that are arrays and only contain the values 0 and 1.

Project>Options>Code Generation>*project*>Optimization>Use system object arrays

-opt_somos

Syntax

`-opt_somos{0|1}`

Parameters

`0` (default)	No system object members are allocated on the stack. This ensures minimal stack usage, but increases the size of the possibly statically allocated) system object.
`1`	Uninitialized candidates are allocated on the stack. This includes members that do not need initialization at allocation time. Stack usage is increased, but the size of the (possibly statically allocated) system object is reduced.
`2`	All candidates are allocated on the stack. Stack usage is increased, but the size of the (possibly statically allocated) system object is reduced.

Scope

Project level.

Description

Determines which system object members to allocate on the stack. Possible candidates include the signal queue, various counters, etc. In a project with multiple systems, the member must be the same size in all systems to be a candidate.

Project>Options>Code Generation>*project*>Optimization>System object members to be stack allocated

-opt_tr

Syntax

`-opt_tr{0|1}`

Parameters

`0` (default)	Optimizes completion transition handling to reduce size. An array of Boolean variables is used to indicate which completion events that are raised. In addition, a counter is used for determining the number of raised completion events, so that the Boolean array is only examined when at least one completion event is raised.
`1`	Optimizes completion transition handling to increase speed. The handling is the same as `-opt_tr0` but in addition, a queue is used. Instead of examining the array of Boolean variables for raised completion events, the queue is searched. The array of Boolean variables is used to ensure that a completion event is not added to the queue multiple times. Overflow will not occur in the queue.

Scope

Project level.

Description

Specifies how to optimize completion transition handling.

Project>Options>Code Generation>*project*>Optimization>Completion transition optimization

-opt_ubabiv

Syntax

`-opt_ubabiv{0|1}`

Parameters

`0`	Bit arrays are not used. This increases application speed.
`1` (default)	Uses bit arrays. This minimizes the size of variable data.

Scope	Project level.
Description	Determines whether bit arrays are used for Boolean internal variables.

Project>Options>Code Generation>*project*>Optimization>Use bit arrays for boolean internal variables

-opt_ubfbev

Syntax

```
-opt_ubfbev{0|1}
```

Parameters

`0`	Bitfields are not used. This increases application speed.
`1` (default)	Uses bitfields. This minimizes the size of variable data.

Scope	Project level.
Description	Determines whether bitfields are used for Boolean external variables.

Project>Options>Code Generation>*project*>Optimization>Use bit fields for boolean external variables

-opt_uso

Syntax

```
-opt_uso{0|1}
```

Parameters

`0` (default)	State offsets are not used instead of fixed state numbers.
`1`	Uses state offsets instead of fixed state numbers. This minimizes RAM usage.

Scope	Project level.
Description	Determines whether state offsets are used instead of fixed state numbers.

Project>Options>Code Generation>*project*>Optimization>Use state offsets

-path

Syntax `-path`*`directory`*

Parameters

`directory`	The output path for all generated project files.

Scope Project level.

Description Specifies the output path for all generated project files. If the path does not exist, it is created. The path can be relative. By default, generated project files are created in the `coder` directory.

See also -*spath*, page 560.

Project>Options>Code Generation>*project*>File Output>Output path

-projectheader

Syntax `-projectheader`*`file`*

Parameters

`file`	The name of the project header file.

Scope Project level.

Description Specifies the name of the header file that contains macros, types, and function prototypes meant for the project. The name used by default is *`Project`*`.h`.

Project>Options>Code Generation>*project*>File Output>Project header file

-projectnamespace

Syntax `-projectnamespace`*`name`*

Parameters

`name`	The name of the project namespace used by generated C++ code.

Scope Project level.

Description

Specifies the C++ namespace for all output for project-related types and functions. By default, the namespace is `""`.

Project>Options>Code Generation>*project*>Code Generation>Project namespace

-projectsource

Syntax

`-projectsource`*`file`*

Parameters

`file`	The name of the project source file.

Scope

Project level.

Description

Specifies the name of the source code file that contains code meant for the project. The name used by default is *`Project.c`*.

Project>Options>Code Generation>*project*>File Output>Project source file

-pssf

Syntax

`-pssf{0|1}`

Parameters

`0` (default)	Separate source files are created for generated code.
`1`	A single source file is created for all generated code.

Scope

Project level.

Description

Determines whether a single source file is created for all generated code. The header files remain separate. Note: A single source file cannot be used for C++ code.

Project>Options>Code Generation>*project*>File Output>Single source file

-pssn

Syntax `-pssn{0|1}`

Parameters

`0`	No symbolic names are generated for states.
`1` (default)	Generates symbolic names for states.

Scope System level.

Description Determines whether the Hierarchical Coder generates symbolic state names.

Project>Options>Code Generation>*system*>Names>Print symbolic state names

-R

Syntax `-R[file]`

Parameters

`file`	The name of the report file.

Scope Project level.

Description Specifies a name for a report file to contain information about the project, option settings, model characteristics, statistics, and a summary of the code generation. If `-R` is specified without an argument, no report file will be created. If this option is not specified at all on the command line, a report with the name `VSCoder.cre` is created.

Project>Options>Code Generation>*project*>File Output>Report file

-recordingbuffersize

Syntax `-recordingbuffersize`*`size`*

Parameters

`size`	The number of elements in the recording buffer.

Scope System level.

Description	Specifies the number of elements in the recording buffer for C-SPYLink.
See also	*-userecordingbuffer*, page 749.

Project>Options>Code Generation>*system*>C-SPYLink>Recording buffer size

-riins

Syntax `-riins{0|1}`

Parameters

`0`	Internal instances cannot be reinitialized.
`1` (default)	Internal instances can be reinitialized.

Scope Project level.

Description Determines whether the internal instances can be reinitialized or not. If they cannot be reinitialized, a reset is required to reach the initial state.

Project>Options>Code Generation>*project*>Memory>Reinitializable internal instance

-S

Syntax `-S`*`file`*

Parameters

`file`	The name of the system source file.

Scope System level.

Description Specifies the name of the source file that contains system-level model definitions. The name used by default is *`System`*`.c`.

Project>Options>Code Generation>*system*>File Output>System source file

-samplingbuffersize

Syntax `-samplingbuffersize`*size*

Parameters

size	The number of elements in the sampling buffer.

Scope System level.

Description Specifies the number of elements in the sampling buffer for C-SPYLink. If the value is too low, you can only see the event that triggered the most recent transition and the states after that microstep. If the value is too high, the target application might run out of memory. This option does not change the behavior of the model.

See also *-usesamplingbuffer*, page 751.

Project>Options>Code Generation>*system*>C-SPYLink>Sampling buffer size

-siss

Syntax `-siss{0|1}`

Parameters

`0` (default)	No explicit initialization of static storage.
`1`	Static storage is explicitly initialized with zero values.

Scope Project level.

Description Determines whether to initialize static storage with zero values (external and internal variables) explicitly. If these initial values are zero, there is no need for an explicit initializer. The option should only be used for compilers that do not perform this initialization as required.

Project>Options>Code Generation>*project*>Code>Explicitly initialize static storage with zero values

-spath

Syntax `-spath`*`directory`*

Parameters

`directory`	The output path for all generated system files.

Scope System level.

Description Specifies the output path for all generated system files. If the path does not exist, it is created. The path can be relative. By default, generated system files are created in the `coder` directory.

See also *-path*, page 555.

Project>Options>Code Generation>*system*>File Output>Output path

-ssewi

Syntax `-ssewi{0|1}`

Parameters

`0`	The start event is not sent automatically.
`1` (default)	The start event is sent automatically.

Scope Project level.

Description Determines whether the start event is sent automatically after `VSInitAll` has been executed. If `-ssewi0` is specified, you must pass the start event into `VSDeduct` manually.

Project>Options>Code Generation>*project*>Code>Send start event when initializing

-suppress_cspylink_common_files

Syntax `-suppress_cspylink_common_files{0|1}`

Parameters

`0` (default)	Disables generation of multiple C-SPYLink files when you are using two or more projects in the same linked image together with C-SPYLink.
`1`	Generates multiple C-SPYLink files when you are using two or more projects in the same linked image together with C-SPYLink.

Scope Project level.

Description Controls how multiple C-SPYLink files are generated when you are using two or more projects in the same linked image together with C-SPYLink.

Project>Options>Code Generation>*project*>C-SPYLink>Suppress C-SPYLink common files

-targetbreakpoints

Syntax `-targetbreakpoints`*`number`*

Parameters

`number`	The number of available breakpoints.

Scope System level.

Description Specifies the number of available breakpoints for C-SPYLink on the target controller. Target breakpoints speed up execution but consume memory. This option does not change the behavior of the model.

Project>Options>Code Generation>*system*>C-SPYLink>Number of state machine breakpoints

-txte

Syntax `-txte{0|1|2|3}`

Parameters

`0` (default)	Includes no text associated with events in the generated code.

`1`	Includes the names of the events in the generated code. This makes it possible to extract the names from the application when it executes on the target. See the documentation for the API functions with suffixes `_Name` and `_NameAbs`.
`2`	Includes the descriptions of the events in the generated code. This makes it possible to extract the descriptions from the application when it executes on the target. See the documentation for the API functions with suffixes `_Expl` and `_ExplAbs`.
`3`	Includes both the names and the descriptions of the events in the generated code.

Scope

System level.

Description

Controls the amount of text associated with events to include in the generated code.

Project>Options>Code Generation>*system*>Names>Event name inclusion

-txtm

Syntax

`-txtm{0|1|2|3}`

Parameters

`0` (default)	Includes no text associated with transition elements in the generated code.
`1`	Includes the names of the transition elements in the generated code. This makes it possible to extract the names from the application when it executes on the target. See the documentation for the API functions with suffixes `_Name` and `_NameAbs`.
`2`	Includes the descriptions of the transition elements in the generated code. This makes it possible to extract the descriptions from the application when it executes on the target. See the documentation for the API functions with suffixes `_Expl` and `_ExplAbs`.
`3`	Includes both the names and the descriptions of the transition elements in the generated code.

Scope

System level.

Description

Controls the amount of text associated with transition elements to include in the generated code.

Project>Options>Code Generation>*system*>Names>State machine name inclusion

-txts

Syntax

`-txts{0|1|2|3}`

Parameters

`0` (default)	Includes no text associated with states in the generated code.
`1`	Includes the names of the states in the generated code. This makes it possible to extract the names from the application when it executes on the target. See the documentation for the API functions with suffixes `_Name` and `_NameAbs`.
`2`	Includes the descriptions of the states in the generated code. This makes it possible to extract the descriptions from the application when it executes on the target. See the documentation for the API functions with suffixes `_Expl` and `_ExplAbs`.
`3`	Includes both the names and the descriptions of the states in the generated code.

Scope

System level.

Description

Controls the amount of text associated with states to include in the generated code.

Project>Options>Code Generation>*system*>Names>State name inclusion

-uselivesamplingbuffer

Syntax

`-uselivesamplingbuffer{0|1}`

Parameters

`0`	Prevents C-SPYLink from reading data from the sampling buffer while the target application is executing.
`1` (default)	Enables C-SPYLink to read data from the sampling buffer while the target application is executing.

Scope

System level.

Description

Determines whether C-SPYLink can read data from the sampling buffer while the target application is executing. The target controller must support live read.

Project>Options>Code Generation>*system*>C-SPYLink>Enable sampling buffer readout

-usepop

Syntax

`-usepop{0|1}`

Parameters

`0`	The Hierarchical Coder uses the output path specified by the `-spath` option for system files.
`1` (default)	The Hierarchical Coder uses the same output path for system files as the path specified for all project files.

Scope

System level.

Description

Determines whether the Hierarchical Coder uses the same output path for system files as the path specified for all project files.

Project>Options>Code Generation>*system*>File Output>Use Project output path

-userecordingbuffer

Syntax

`-userecordingbuffer{0|1}`

Parameters

`0`	Disables the recording buffer.
`1` (default)	Enables the recording buffer.

Scope

System level.

Description

Determines whether to use a recording buffer to make it possible to make recordings (execution logs) at almost full speed. Enabling the buffer also makes it possible to display sampling backups. Use the option `-recordingbuffersize` to set the size of the buffer.

See also

-recordingbuffersize, page 734.

Project>Options>Code Generation>*system*>C-SPYLink>Enable recording buffer

-usesamplingbuffer

Syntax

`-usesamplingbuffer{0|1}`

Parameters

`0` (default)	Disables on-target sampling buffers for a single macrostep.
`1`	Enables on-target sampling buffers for a single macrostep.

Scope

System level.

Description

Controls on-target sampling buffers for a single macro step. If you specify `-usesamplingbuffer1`, C-SPYLink can extract large amounts of debug information from your model. This causes an increase in code size and a small reduction in execution speed. If sequence recording is used, the speed reduction will be larger. Use the option `-samplingbuffersize` to set the size of the buffer.

See also

-samplingbuffersize, page 735.

Project>Options>Code Generation>*system*>C-SPYLink>Enable sampling buffer

-V

Syntax

`-V`*`system`*

Parameters

`system`	The name of a system.

Scope

System level.

Description

Specifies the system that the subsequent system options on the command line apply to. System options that are specified before a `-V` option apply to all systems.

This option is not needed in the graphical interface.

-variant

Syntax `-variant`*`name`*

Parameters

`name`	The name of the variant.

Scope Project level.

Description Specifies which variant to generate code for. By default, the Coder generates code for the complete model.

See also *Using variants and features*, page 217.

Use the **Variant** toolbar.

-warnings_affect_exit_code

Syntax `-warnings_affect_exit_code{0|1}`

Parameters

`0` (default)	Warnings generate a zero exit code.
`1`	Warnings generate a non-zero exit code.

Scope Project level.

Description By default, the exit code is not affected by warnings, because only errors produce a non-zero exit code. This option determines whether warnings also generate a non-zero exit code.

Project>Options>Code Generation>*project*>Configuration>Warnings affect exit code

-warnings_are_errors

Syntax `-warnings_are_errors{0|1}`

Parameters

`0` (default)	Warnings are treated like warnings.
`1`	All warnings are reclassified as errors.

Scope

Project level.

Description

Determines whether all warnings are reclassified as errors. If the Hierarchical Coder encounters an error, no code is generated.

Project>Options>Code Generation>*project*>Configuration>Treat warnings as errors

-width_babiv

Syntax

`-width_babiv{0|1|2|3}`

Parameters

`0` (default)	Informs the HCoder that Boolean internal variables encoded as bit arrays are 8 bits.
`1`	Informs the HCoder that Boolean internal variables encoded as bit arrays are 16 bits.
`2`	Informs the HCoder that Boolean internal variables encoded as bit arrays are 32 bits.
`3`	Informs the HCoder that Boolean internal variables encoded as bit arrays are 64 bits.

Scope

Project level.

Description

Specifies the data width for Boolean internal variables encoded as bit arrays.

Project>Options>Code Generation>*project*>Optimization>Width of type for boolean internal variables bit arrays

Adaptive API code generation

- Introduction to the Adaptive API code generation
- Using the Adaptive API

Before you read about Adaptive API code generation, you should be familiar with code generation in general. See *Code generation*, page 457.

Introduction to the Adaptive API code generation

Learn more about:

BRIEFLY ABOUT ADAPTIVE API CODE GENERATION

Code for the Adaptive API can only be generated by the Classic Coder.

Choose between two fundamentally different variants of source code output:

- *Table-based code* (C or C++) for maximum compactness. The state machine logic is encoded in compact tables.
- *Readable code* (C, C#, or Java), a plain representation of the state machine logic, based on `switch` and `if` statements.

  The readable code variant 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, readable code is generally preferable.

Both the readable code representation and the table-based representation of the state machine logic have their strengths and weaknesses. In particular:

- Readable code can be inspected and reviewed, with an easily understood mapping from state machine model to code.

- The readable representation is a straightforward translation into plain C, C#, or Java code. In contrast, the table-based code consists of tables that represent the state machine logic plus code to interpret the tables. This means that readable code will generally be faster.
- Table-based code is more compact. The size added by calling action functions and guard expressions/assignments is also very low.
- The readable code calls actions and guards/assignments in place, which makes the total code size much more dependent on the state machine model. For example, adding a call to an action function on a transition will add an explicit function call in the generated code. In this respect, the readable code is much closer to what user-written code would look like.

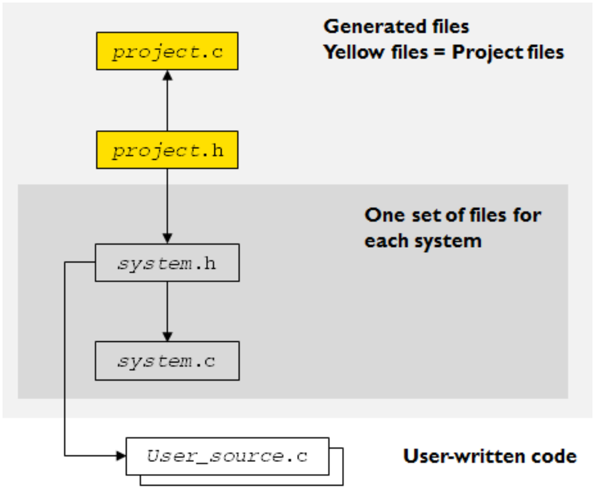
FILE STRUCTURE FOR ADAPTIVE API CODE

During the code generation phase, these sets of files are generated:

- Project-specific files
- System-specific files

For C# and Java, only source files are generated, because these languages do not have the header file concept. Enumerations (both predefined and user-defined) are generated in separate source files.

This figure shows the Coder-generated files and Adaptive API files to be used in your compiler project—for example, in the IAR Embedded Workbench IDE:



In the figure, the dark gray area represents the source and header files that are part of the API. The arrows in the figure indicate how the header files are included in other files. `System` stands for the system name. The generated API files are the same for C and C++ code (with the appropriate filename extension) and for table-based and readable code.

For a list of generated files, see *Coder-generated files for Adaptive API code*, page 589.

ADAPTIVE API TABLE-BASED CODE WITH C++

The Coder can generate C++ code for the Adaptive API. The generated C++ files conform to the Embedded C++ standard.

C++ code generated for the Adaptive API uses C++ to expose its external interface, but uses C internally to keep the generated code compact and efficient.

Generating C++ code has the following advantages:

- User-written code that interfaces to the generated code can interact with a class that uses C++ language features such as the keyword `private` to protect its members from accidental and/or prohibited access.
- To many developers, exposing a C++ interface is more elegant than exposing a C interface.
- In your user-written code you can create any number of instances of the Visual State system, and the instances can be allocated statically or dynamically at the same time. This feature is not available in the Adaptive and Uniform APIs when generating C code. In addition, instances do not share any internal data memory (do not include external variables) and therefore it is easier to enable thread safety in your application.

The performance of C++ code generated for the Adaptive API is about similar to the performance of C code generated for the Uniform API.

File structure for Adaptive API table-based C++ code

The file structure to be used in your compiler project—for example, in the IAR Embedded Workbench IDE—is the same for all Adaptive API code, see *File structure for Adaptive API code*, page 570.

Using the default Coder options, the generated C source files have the filename extension `c`, and the generated C++ source files have the filename extension `cpp`. You can change these extensions in the **Classic Coder Options** dialog box.

ADAPTIVE API READABLE CODE

With the readable code, both the API for calling the generated code and the set of generated files is simplified compared to table-based code. The readable code API supports C, C#, and Java but not C++, and the resulting application cannot be debugged using RealLink. However, readable C code can be debugged with the C-SPY Simulator or a hardware debugging system in the IAR Embedded Workbench IDE, using C-SPYLink.

File structure for Adaptive API readable code

The file structure to be used in your compiler project—for example, in the IAR Embedded Workbench IDE—is the same for all Adaptive API code, see *File structure for Adaptive API code*, page 570.

Using the Adaptive API

What do you want to do?

- *Getting started generating code for the Adaptive API*, page 572
- *Generating code for an API*, page 572
- *Setting up the file structure for Adaptive API*, page 574
- *Using the API*, page 574
- *Using the Adaptive API for table-based code and C++*, page 582
- *Converting table-based C applications to C++ code*, page 584

See also:

- *Introduction to code generation, the Coders, and the APIs*, page 457
- *Adaptive API code generation*, page 569
- *Descriptions of the Adaptive API functions*, page 592
- *Classic Coder command line options*, page 701, for information about how to start code generation from the command line

GETTING STARTED GENERATING CODE FOR THE ADAPTIVE API

1 *Generating code for an API*, page 572.

2 *Setting up the file structure for Adaptive API*, page 574.

3 *Using the API*, page 574.

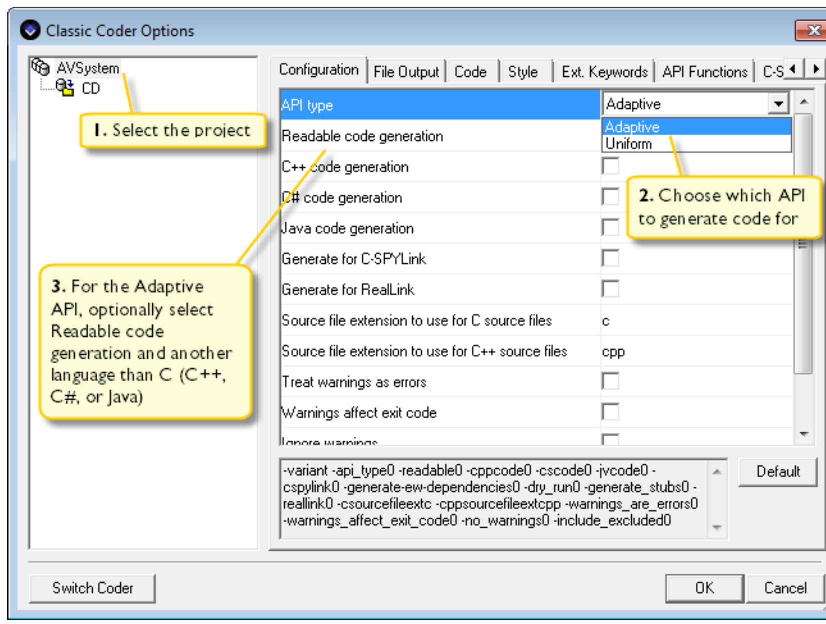
If you want C++ support, see also *Using the Adaptive API for table-based code and C++*, page 582.

GENERATING CODE FOR AN API

1 In the Navigator, open your workspace file.

2 Choose **Project>Options>Code generation** to open the **Classic Coder Options** dialog box.

3 In the left pane, select the project and make the required settings on the **Configuration** page:



Make your settings:

- **API type**: choose Adaptive or Uniform
- **Readable code generation**: generates readable code instead of table-based code (requires the Adaptive API)
- **C++ code generation**: generates C++ code (requires the Adaptive API).
- **C# code generation**: generates C# code instead of C or C++ (requires the Adaptive API).
- **Java code generation**: generates Java code instead of C or C++ (requires the Adaptive API).

If any of the options are not enabled, right-click the option for information about how to enable it.

For reference information about the dialog box, see *Classic Coder Options dialog box*, page 674.

4 Click **OK** when finished.

5 Choose **Project>Code generate** to generate code for the project.

Code generation starts, and progress is displayed in the **Output** window.

By default, Coder-generated files are located in the `Coder` directory in your project directory (where the project file is located). You can specify another file output directory on the **File Output** page in the **Classic Coder Options** dialog box.

SETTING UP THE FILE STRUCTURE FOR ADAPTIVE API

1 Include the Adaptive API header file *`System`*`.h` in your source code (*`System`* reflects the name of the Visual State system.)

2 Write code to act as an interface to the Adaptive API:

- Call all the required initialization functions, see *Calling initialization functions*, page 575.
- Call the Adaptive API functions in sequences as described in *Calling event deduction functions*, page 576, *Performing an event inquiry*, page 576, *Retrieving names and descriptions*, page 577, and *Retrieving and setting states*, page 578.

3 Implement the action functions that are needed by your application.

4 Include the following source files in a make file:

- The Adaptive API source file *`System`*`.c`.
- The project source file *`Project`*`.c`. (*`Project`* stands for the name of your project.)
- Your source file.

5 Add your compiler and linker commands to the make file.

USING THE API

You use functions in the Adaptive API (table-based or readable) and the Uniform API in the same way, except for a few differences. In case of a difference, this is clearly stated.

The API functions are divided in groups of related functionality and typically this is what you must consider doing:

- *Connecting and disconnecting functions (Uniform API only)*, page 575
- *Calling initialization functions*, page 575
- *Calling event deduction functions*, page 576
- *Performing an event inquiry*, page 576
- *Retrieving names and descriptions*, page 577

For information about the various API functions, see:

Note: For readable C# and Java code, the API functions are not prefixed with the system name or project name, as the system name is used as the class name in these cases.

Connecting and disconnecting functions (Uniform API only)

When using the Uniform API, it is necessary to acquire a system context (a handle) before calling any other API functions. Such a context is acquired by calling the connecting function `SMP_Connect`. Use the acquired system context for all the subsequent calls to API functions that must operate on this particular system.

To release the system context:

1. Call the API function `SMP_Free` with a system context. After a call to `SMP_Free`, the system context can no longer be used.
2. When a call has been made to `SMP_Connect`, a system is said to be loaded. When a call has been made to `SMP_Free`, a system is said to be unloaded.

Calling initialization functions

Calling the initialization functions is required to ensure proper initialization of your project.

To ensure proper initialization:

1. From the user-written code, call the API function for initialization, which means:
   - For Adaptive API readable code or if you enabled the Coder option `-vsintiall`: call the API function *`Systemname`*`VSInitAll`.
   - For Adaptive API table-based code: call the API function *`Systemname`*`SEM_InitAll`.

- For Uniform API table-based code: call the API function *Systemname*SMP_InitAll.

  If you use this function for the Uniform API, the function will also connect the system which means that you do not also need to call SMP_Connect.

Calling the *InitAll function takes care of most of the initialization you need.

2 If you have any project-external variables, you must call an initialization function somewhere from your code for those if you have set **External variable initialization** to **Both** in the **Classic Coder Options** dialog box, which means:

- For Adaptive API table-based code: call the generated function *Project*SEM_InitPrjExternalVariables.
- For Adaptive API readable code: call the generated function *Project*VSInitPrjExternalVariables.
- For Uniform API table-based code: call the generated function *Project*SEM_InitPrjExternalVariables.

  If you use this function for the Uniform API, the function will also connect the system so you do not also need to call SMP_Connect.

Calling event deduction functions

Event deduction is also referred to as macrosteps in Visual State. See *Runtime behavior—macrosteps and microsteps*, page 122.

To perform event deduction in the application:

1 Your code must somehow obtain an event from the environment. This event must be mapped to a Visual State event (symbolic event names are generated in the *System*.h file if the Coder option -sne has been set). The first event used in a macrostep following the initialization procedure must be the Visual State reset event SE_RESET.

2 Call SystemVSDeduct, the API function with the Visual State event.

If the event has parameters, supply these as additional parameters to the function.

Performing an event inquiry

The API provides functions that can determine active events, in other words, which events will cause the state configuration to change. By default, these functions are not enabled, but must be enabled by setting the appropriate Coder options.

To perform an event inquiry:

1 For table-based code, call the API function SXX_Inquiry (where SXX is SEM for the Adaptive API, but SMP for the Uniform API). For readable code, call the API function VSInquiry.

The next step is only applicable to table-based code.

2 Call the API function SXX_GetInput repeatedly until all active events have been retrieved.

As an alternative to calling SXX_GetInput multiple times, call the API function SXX_GetInputAll once, which returns all active events in a buffer.

Retrieving names and descriptions

The API provides text functions by which it is possible to get the name and description of a specified event, state, or action function. By default, these functions are not enabled, so you must enable them by setting the appropriate Coder options.

To retrieve the name of an element:

1 For readable code, or if you have enabled the Coder option -vs*elementname*, perform this step (otherwise skip this step and go directly to step 2):

- Call the API function VSElementName. This returns a pointer to the internal representation of the name.
- Now you can use the name after receiving the pointer.

  Do not perform the next step.

2 Call the API function SXX_Name or SXX_NameAbs (where SXX is SEM for the Adaptive API, but SMP for the Uniform API). The former copies the name to a specified buffer, while the latter returns a pointer to the internal representation of the name.

Now you can use the name after receiving it in the buffer (using SXX_NAME) or the pointer (using SXX_NameAbs).

To retrieve the description of an element:

1 For readable code, or if you have enabled the Coder option -vs*elementexpl*, perform this step (otherwise skip this step and go directly to step 2):

- Call the API function VSElementExpl. This returns a pointer to the internal representation of the description.
- Now you can use the explanation after receiving the pointer.

  Do not perform the next step.

2 Call the API function SXX_Expl or SXX_ExplAbs (where SXX is SEM for the Adaptive API, but SMP for the Uniform API). The former copies the description to a specified buffer, while the latter returns a pointer to the internal representation of the description.

Now you can use the explanation after receiving it in the buffer (using SXX_Expl) or the pointer (using SXX_ExplAbs).

Retrieving and setting states

The API provides functions by which it is possible to retrieve information on the internal state configuration, and to force the states in the internal state configuration into a specific state. By default, these functions are not enabled, but must be enabled by setting the appropriate Coder options.

Note: Each state is owned by one specific parent state machine. This means that states cannot have the same state index number across state machines (state index numbers are unique).

To retrieve information on the internal state configuration:

1 Call the API function with a state index number to determine the parent state machine, which means:

- For Adaptive API table-based code: use the function `SEM_Machine`
- For Adaptive API readable code: use the function `VSMachine`
- For Uniform API table-based code: use the function `SMP_Machine`.

2 Call the API function with a state machine index number to determine the current state of the state machine, which means:

- For Adaptive API table-based code: use the function `SEM_State`
- For Adaptive API readable code: use the function `VSState`
- For Uniform API table-based code: use the function `SMP_State`.

To force a state in the internal state configuration into a specific state:

1 Find the state you want to force some state machine into. This state might be one you have stored just before running out of power.

2 Call the API function with a state index number to force the parent state machine into this new state, which means:

- For Adaptive API table-based code: use the function `SEM_ForceState`
- For Adaptive API readable code: use the function `VSForceState`
- For Uniform API table-based code: use the function `SMP_ForceState`.

This function should primarily be used for restoring a previous state configuration obtained by calls to `SEM_State|VSState|SMP_State`.

Forcing a single state machine into a specified state might result in an *illegal* state configuration, that is a configuration which would not otherwise be reachable. Thus, it will not be covered by a verification with the Verificator. In general, use this function cautiously.

Managing instances

The API is capable of handling multiple instances of the same Visual State system, see *Reuse of design using system instances*, page 126. The system exists only in one location in memory. The only information multiplied is variables for storing the current state configuration and the internal variables.

For table-based code:	It is illegal to change instances when states might be changing, in other words, in the middle of a call to `VSDeduct`. Likewise, when `SXX_ForceState` is used (where `SXX` is `SEM` for the Adaptive API, but `SMP` for the Uniform API), ensure that the correct instance is updated.
For readable code:	If your system has more than one instance specified in the Designer, some functions will automatically be prepared for using instances. This applies to: `VSDeductInstance`, `VSForceStateInstance`, `VSInquiryInstance`, `VSStateInstance`. These functions work in the same way as the ordinary functions, (without the `Instance` postfix), and they all take an extra argument that indicates which instance to work with.

To use multiple instances:

1. Specify the number of instances in the Designer, see *Creating multiple system instances*, page 235.
2. Call the API function, which means:
   - For Adaptive API readable code or if you enable the `-vsinitall` Coder option: use the function *`System`*`VSInitAll`
   - For Adaptive API table-based code: use the function `SEM_InitAll`
   - For Uniform API table-based code: use the function `SMP_InitAll`.

   If needed, the function will in its turn initialize instances.
3. Each time an event deduction is performed, set the correct instance, which means:
   - For table-based code: use the function `SXX_SetInstance`
   - For readable code: the correct instance simply need to be specified when you call, for example `VSDeductInstance`.

If these guidelines are followed, the API can handle multiple instances in a pseudo-parallel manner without any reduction in performance.

For some example code on how to use `SEM_SetInstance`, see *SEM_SetInstance*, page 608.

Managing internal variables

For table-based code: Internal variables are defined at system level in the Designer and you specify whether the variables should be initialized by definition or by an initialization function. Internal variables will be placed in the *System*.c file.

If the -iiv1 Coder option is set, the Coder will generate the variable initialization function SXX_InitInternalVariables (where SXX is SEM for the Adaptive API, but SMP for the Uniform API). The function is placed in the *System*.c file.

For readable code: Internal variables are defined at system level in the Designer and you specify whether the variables should be initialized by definition or by an initialization function. Internal variables will be placed in the *System*.c file.

If the -iiv1 Coder option is set, the Coder will generate the variable initialization function VSInitInternalVariables. The function is placed in the *System*.c.

Managing external variables

External variables are defined at project level or system level in the Designer and you specify whether the variables should be initialized by definition or by an initialization function. If you have any project-external variables, and you have set the Coder option -iev so that you get a function for initializing external variables, you must call that generated function:

For table-based code: *InitPrjExternalVariables

External variables are by default declared in the *System*.h file and will be placed in the *System*.c file. But in the **Classic Coder Options** dialog box you can choose other destination files. If the Coder option -iev1 is set, the Coder will generate the external variable initialization function SXX_InitExternalVariables (where SXX is SEM for the Adaptive API, but SMP for the Uniform API) for initializing system-external variables. The function is placed in the same file as the variables.

For readable code: `*VSInitPrjExternalVariables`

External variables are by default declared in the *System*.h file and will be placed in the *System*.c file. But in the **Classic Coder Options** dialog box you can choose other destination files. If the Coder option `-iev1` is set, the Coder will generate the external variable initialization function `VSInitExternalVariables` for initializing system-external variables. The function is placed in the same file as the variables.

Managing constants

Constants can be defined at project level or system level in the Designer.

- Constants defined at project level will be defined in the *Project*.h file.
- Constants defined at system level will be defined in the *System*.h file.

All constants will be defined as C constants.

Managing enumerations

Enumerations can only be defined in transition element files at System level in the Designer.

Enumerations will always be generated in a header file with the same name as the enumeration. For C and C++, the C enumeration format is used.

Managing signals

Signals are handled internally. An API function sends signals to the signal queue and empties the signal queue, which means that you should use the function `VSDeduct`.

If signals are used, you must enable the signal queue by specifying a signal queue size. The signal queue size must be large enough to contain the largest number of signals that can be caused by an event.

For table-based code, the `SEM_InitSignalQueue` function (`SMP_InitSignalQueue` for the Uniform API) initializes the signal queue and will be enabled by the Coder if the signal queue size is larger than zero.

For the Adaptive API (readable), or if you enabled the `-vsinitall` Coder option: *System*`VSInitAll` will automatically call the appropriate function to initialize the signal queue.

For the Adaptive API (table-based):	`SEM_InitAll` will automatically call `SEM_InitSignalQueue` if necessary.
For the Uniform API:	*`System`*`SMP_InitAll` will also call *`System`*`SMP_InitSignalQueue` as part of initializing the system.

Managing event arguments

When an event takes arguments, the arguments must be given together with the event as arguments to the *`System`*`VSDeduct` function.

For more information, see *VSDeduct*, page 612.

USING THE ADAPTIVE API FOR TABLE-BASED CODE AND C++

The Coder does not instantiate objects of the generated system class (named `VS_SYSTEM`). Therefore, you must instantiate objects of the system class in your own files (user-written code).

In contrast to a standard Adaptive API application, any number of objects of the system class can be instantiated, just as is the case for ordinary classes. Because the objects do not share any internal data memory (they do not include external variables), two different objects of the system class can be accessed simultaneously from two different threads, provided that all functions are reentrant, and external variables are not modified.

When generating C++ code, you must interface to member functions of the generated system class instead of global functions. For every API function that you must call for a C application, you must call a corresponding member function (having the same name) of the generated class.

Instances in C++ API code

Each Visual State system consists of one or more instances with exactly one instance being active at any point in time, see *Reuse of design using system instances*, page 126. Such instances are called *internal instances* and they have these characteristics:

- The number of internal instances is fixed for the system at the time of code generation. You can specify the number of internal instances in the Designer in the **Edit Systems** dialog box. See *Creating multiple system instances*, page 235.
- Only one internal instance may be active at a time because internal instances share internal data memory.

Internal instances should not be mistaken for instances (objects) of the generated class, which are called *external instances* and they have these characteristics:

- External instances can be instantiated any number of times, either statically, on the stack, or in the heap.
- Multiple external instances may be manipulated at the same time because external instances do not share internal data memory (do not include external variables).

Both types of instances may be referred to as just instances when the type of instance clearly appears from the context.

Internal variables in C++ API code

Internal variables are part of the generated class as private member variables. Consequently they can only be initialized by an initialization function.

External variables in C++ API code

External variables are not part of the generated class, but are generated as statically allocated variables, in the same way as for a C application. Therefore, all external instances of the generated class share the same set of external variables.

If two external instances manipulate an external variable from two different threads, you must synchronize the access to that variable.

Constants in C++ API code

Constants are not part of the generated class, but are generated in the same way as for a C application.

Enumerations in C++ API code

Enumerations are not part of the generated class, but are generated in the same way as for a C application.

Signals in C++ API code

Signals are handled internally, in the same way as for a C application. Note that every external instance has its own signal queue, while internal instances share a single signal queue.

Event parameters in C++ API code

Event parameters are handled in the same way as for a C application. The Coder will always generate a member function `SEM_Deduct` for the generated class, independently of the existence of event parameters.

CONVERTING TABLE-BASED C APPLICATIONS TO C++ CODE

If you have an existing C application, you can easily modify your files for C++ code generation.

For each call to an API function, prefix the function name with the name of the object that you instantiate, followed by a period (this is the syntax for calling a member function of a class).

For example, a call to `SEM_InitAll` that has the form `SEM_InitAll()` should be replaced by `System.SEM_InitAll()` (in this example, it is assumed that the object is named `System`).

Uniform API code generation

Before you read about Uniform API code generation, you should be familiar with code generation in general. See *Code generation*, page 457.

Introduction to the Uniform API code generation

Learn more about:

BRIEFLY ABOUT UNIFORM API CODE GENERATION

Code for the Expert API can only be generated by the Classic Coder.

The Uniform API will be generated in two files: `project.c` and `project.h`, where `project` reflects the name of your project file.

Most of the functions in the Uniform API have SMP as prefix, and they take a pointer to a system context as a parameter to determine the system to operate on. This means that the API can operate on projects that contain multiple systems.

For information about the functions, see *Descriptions of the Uniform API functions*, page 638.

Projects with multiple systems and reentrancy

Because all SMP functions are reentrant and are passed with a system context as a parameter, multiple operating system tasks may operate on different systems at the same time. Because of the principle of reentrancy, simultaneous calls made to the same API function will not cause problems as long as none of the simultaneous calls use the same system contexts as parameters to the function in question. Likewise, simultaneous calls to different API functions are supported. In general, simultaneous calls to API functions with different system contexts are supported. For example, event deductions may be in progress in different operating system tasks at the same time, all retrieving action expressions from the `SMP_GetOutput` function.

This is an example of how system contexts are used; a system context pointer variable is defined for each system:

```
SEM_CONTEXT *pSystemContext;
```

The system context pointer is assigned by calling the initialization function:

```
SystemSMP_InitAll(&pSystemContext, &VSSystem);
if (CC != SES_OKAY)
  exit (CC);
```

The system context pointer used in an event deduction (macrostep):

```
SEM_EVENT_TYPE EventNo;
SEM_ACTION_EXPRESSION_TYPE ActionExp;

. . .
CC = SystemVSDeduct (pSystemContext, EventNo);
if (CC != SES_OKAY && CC != SES_FOUND)
  exit (CC);
```

Reentrancy of SMP functions depends on the compiler used. Thus, the compiler used for compilation of API source files must also support reentrancy, because some of the API functions use local stack variables. If the compiler does not support reentrancy, local stack variables may be stored in fixed memory locations, and different operating system tasks controlling different systems might access the same variable space simultaneously which will result in unpredictable behavior.

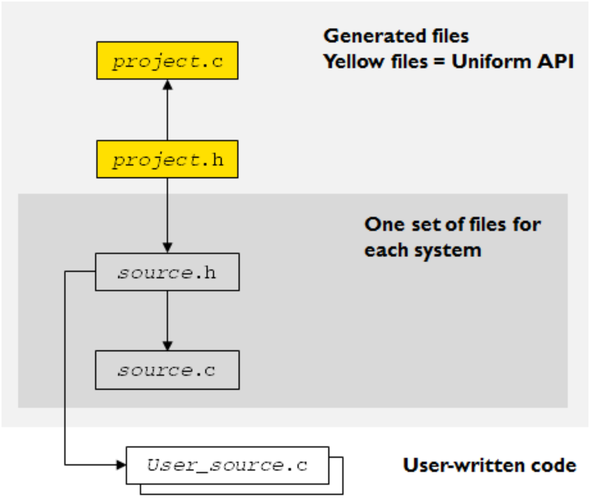
UNIFORM API CODE

During the code generation phase, these sets of files are generated:

- Project-specific files
- Project-specific API files
- System-specific files

File structure for Uniform API table-based code

This figure shows the Coder-generated files and Uniform API files to be used in your compiler project—for example, in the IAR Embedded Workbench IDE—for table-based code:



In the figure, the rectangle in yellow represents the header files that are part of the API. The arrows in the figure indicate how the header files are included in the source files. There must be a user-written source file for each system. As can be seen in the figure, each such file must include all project header files and all system header files for a specific system.

For a list of generated files, see *Coder-generated source files for the Uniform API*, page 635.

Using the Uniform API

What do you want to do?

- *Getting started generating code for the Uniform API*, page 588
- *Setting up the file structure for the Uniform API*, page 588

See also:

- *Introduction to code generation, the Coders, and the APIs*, page 457
- *Introduction to the Uniform API code generation*, page 585

- *Adaptive API reference information*, page 589
- *Classic Coder command line options*, page 701, for information about how to start code generation from the command line

GETTING STARTED GENERATING CODE FOR THE UNIFORM API

1. *Generating code for an API*, page 572.

   Note: This task is described in the chapter *Adaptive API code generation*.

2. *Setting up the file structure for the Uniform API*, page 588.
3. *Using the API*, page 574.

   Note: This task is described in the chapter *Adaptive API code generation*.

SETTING UP THE FILE STRUCTURE FOR THE UNIFORM API

1. Include these header files in all your source files:
   - The Uniform API header file `project.h`.
   - The Coder-generated header files for the specific system, `source.h`.
2. Write code that interfaces to the Uniform API:
   - Call one of the connecting functions, see *Connecting and disconnecting functions (Uniform API only)*, page 575.
   - Call all the required initialization functions, see *Calling initialization functions*, page 575.
   - Call the Uniform API functions in sequences as described in *Calling event deduction functions*, page 576, *Performing an event inquiry*, page 576, *Retrieving names and descriptions*, page 577, and *Retrieving and setting states*, page 578.
   - Call the disconnecting function `SMP_Free`, see *Connecting and disconnecting functions (Uniform API only)*, page 575.
3. Include the following source files in a make file:
   - The Uniform API source file `project.c`.
   - The Coder-generated system source files, `source.c`.
   - Your source files.
4. Add your compiler and linker commands to the make file.

Adaptive API reference information

- Coder-generated source files for the Adaptive API
- Summary of the Adaptive API functions
- Descriptions of the Adaptive API functions
- Adaptive API return codes

Coder-generated source files for the Adaptive API

Declarations for all Adaptive API functions are located in the API header file *System*.h.

Unless otherwise stated, the portability of the Adaptive API functions is Standard C compliant (Embedded C++ in the case of C++ code generation).

Learn more about:

- Coder-generated files for Adaptive API code

CODER-GENERATED FILES FOR ADAPTIVE API CODE

During the code generation phase, these sets of files are generated:

- Project-specific files
- System-specific files

These are the project-specific files:

Project.h	Contains the declarations of all project-related types, and external variables that are defined at project level and shared for all systems.
Project.c	Contains the definitions of all external variables that are defined at project level and shared for all systems.

Project stands for the project name.

These are the system-specific files:

System.c	Contains the core model logic of the system (primarily transitions).
System.h	Header files for *System*.c. Contains all relevant types and macros for the system.

System stands for the prefix used by the code generator, to distinguish files from different systems. The default prefix is the system name, but you can change it in the **Classic 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.

For readable C# and Java code, no header files are generated. The filename extensions for the source files are .cs and .java, respectively, and cannot be changed. To handle action functions, the interface source files I*Systemname*ActionHandler and I*Projectname*ActionHandler are generated. Any enumerations are generated in separate files, named after the enumeration. The predefined enumerations IdentifierType and VSResult are always generated in their own separate files.

Summary of the Adaptive API functions

This table summarizes the Adaptive API functions:

Adaptive API function	Description
SEM_Expl	Gets the ASCII description of a specified identifier.
SEM_ExplAbs	Gets the absolute address of an ASCII description of a specified identifier.
SEM_ForceState	Forces the internal state configuration into a specified state.
SEM_GetInput	Finds events that can trigger transitions or derive action expressions from the current state.
SEM_GetInputAll	Finds all events that can trigger transitions or derive action expressions from the current state.
SEM_Init	Initializes the system and must be called before any other functions are called.
SEM_InitAll	Wraps all initialization functions and calls them in order. This is the recommended way to initialize a system.

Table 29: Summary of the Adaptive API functions

Adaptive API function	Description
SEM_InitExternalVariables	Initializes the external variables in the system.
SEM_InitInstances	Initializes a number of instances of a system.
SEM_InitInternalVariables	Initializes the internal variables in the system.
SEM_InitSignalQueue	Initializes the signal queue in a system.
SEM_Inquiry	Prepares for finding events that can trigger changes in the current state.
SEM_Machine	Returns the state machine index of a specified state.
SEM_Name	Gets the ASCII name of a specified identifier.
SEM_NameAbs	Gets a pointer to the ASCII name of a specified identifier.
SEM_SetInstance	Sets the currently active instance of the system.
SEM_SignalQueueInfo	Returns information about the signal queue.
SEM_State	Returns the current state of a specified state machine.
SEM_StateAll	Returns the active state of all state machines.
VSDeduct	Deduces all the relevant action expressions on the basis of the given event, the internal current state vector, and the transitions in the Visual State system.
VSDeductInstance	Deduces all the relevant action expressions on the basis of the given event, the internal current state vector, and the transitions in the Visual State system for the given instance.
VSElementExpl	Gets the pointer to the explanation for the specified identifier.
VSElementName	Gets the pointer to the ASCII name of the specified identifier.
VSForceState	Forces the internal state configuration to the specified state.
VSForceStateInstance	Forces the internal state configuration to the specified state for the given instance of a system.
SystemVSGetCurrentStateTree	Copies the strings representing the current state tree into the buffer.
*System*VSGetMaxCurrentStateTree	Returns the needed size for VSGetCurrentStateTree buffer.
VSInitAll	Wraps all initialization functions.
VSInitExternalVariables	Initializes the external variables in the system and must be called together with the VSInitAll function.

Table 29: Summary of the Adaptive API functions (Continued)

Adaptive API function	Description
VSInitInternalVariables	Initializes the internal variables in the system and must be called together with the VSInitAll function.
VSInquiry	Finds events that can trigger transitions or derive action expressions from the current state.
VSInquiryInstance	Finds events that can trigger transitions or derive action expressions from the current state configuration for the given instance.
VSMachine	Returns the state machine index of the specified state.
VSState	Returns the current state of the specified state machine.
VSStateAll	Returns the active state of all state machines.
VSStateAllInstance	Returns the active state of all state machines for the given instance of the system.
VSStateInstance	Returns the current state of the specified state machine for the specified instance.

Table 29: Summary of the Adaptive API functions (Continued)

Descriptions of the Adaptive API functions

The following pages give detailed reference information about each Adaptive API function. The syntax descriptions and examples apply to C/C++, for C# and Java they are slightly different. For the exact syntax for C# and Java API functions, inspect the generated code.

SEM_Expl

Syntax

```
unsigned char SEM_Expl (unsigned char IdentType,
                        SEM_EXPLANATION_TYPE IdentNo,
                        char *Text,
                        unsigned short MaxSize)
```

Defined in

`SystemSEMLibB.c`

For use with

Table-based code

Description

This deprecated function is provided for backward compatibility and should not be used. Instead use the function `VSElementExpl`, see *VSElementExpl*, page 616.

This function gets the ASCII description of the specified identifier.

The function must be enabled by the Coder command line option `-semexpl1` or the corresponding GUI option.

Parameters

`IdentType`	Must contain the type of the identifier, `EVENT_TYPE` or `STATE_TYPE`. Note that `ACTION_TYPE` is deprecated because there is no advantage to using it here.
`IdentNo`	Must contain the index number of the identifier.
`Text`	Must contain a pointer to a text string buffer. If the function terminates successfully, the text string contains the name of the specified identifier.
`MaxSize`	Specifies the maximum length of the text including the `NULL` termination character.

Return value

See:

SES_RANGE_ERR, page 633

SES_TEXT_TOO_LONG, page 634

SES_TYPE_ERR, page 634

SES_OKAY, page 633

Example

See *SEM_GetInput*, page 595.

SEM_ExplAbs

Syntax

```
unsigned char SEM_ExplAbs (unsigned char IdentType,
                           SEM_EXPLANATION_TYPE IdentNo,
                           char **Text)
```

Defined in

System`SEMLibB.c`

For use with

Table-based code

Description

This deprecated function is provided for backward compatibility and should not be used. Instead use the function `VSElementExpl`, see *VSElementExpl*, page 616.

This function gets the absolute address of an ASCII description of the specified identifier.

The function must be enabled by the Coder command line option `-semexplabs1` or the corresponding GUI option.

Parameters

`IdentType`	Must contain the type of the identifier, `EVENT_TYPE` or `STATE_TYPE`. Note that `ACTION_TYPE` is deprecated because there is no advantage to using it here.
`IdentNo`	Must contain the index number of the identifier.
`Text`	Must be a pointer to a `char *`. If the function terminates successfully, the pointer contains the absolute address of the name of the specified identifier.

Return value

See:

SES_RANGE_ERR, page 633

SES_TYPE_ERR, page 634

SES_OKAY, page 633

Example

See *SEM_GetInputAll*, page 597.

SEM_ForceState

Syntax

`unsigned char SEM_ForceState (SEM_STATE_TYPE StateNo)`

Defined in

System`SEMLibB.c`

For use with

Table-based code

Description

This function is used for forcing the internal state configuration into the specified state. This is useful if you want to reestablish the internal state configuration after a power failure of the target system. Before calling this function the first time after a power failure, the `SEM_InitAll` function should be called to initialize the other internal variables of the system.

The state configuration established by calling `SEM_ForceState` must have been stored in EEPROM before the power failure.

Note: This function should be used with caution. The internal state configuration could be forced to a configuration that is not reachable by executing the model itself.

The function must be enabled by the Coder command line option `-semforcestate1` or the corresponding GUI option.

Parameters

`StateNo`	Contains the state index number.

Return value

See:

SES_RANGE_ERR, page 633

SES_OKAY, page 633

Example

```
/*
  * This function should only be called after a power failure
  * to reestablish the internal state variables.
  */
void PowerUp (void)
{
  SEM_STATE_TYPE StateNo;
  SEM_STATE_MACHINE_TYPE i;

  /* Initialize the VS System. */
  SEM_InitAll ();

  for (i = 0; i < VS_NOF_STATE_MACHINES; i++)
  {
    /* Get state configuration from EEPROM. */
    EEPROMState (i, &StateNo);
    if (SEM_ForceState (StateNo) != SES_OKAY)
      ErrorHandling ();
  }
}
```

See also

-*snm*, page 737.

SEM_GetInput

Syntax

```
unsigned char SEM_GetInput (
  SEM_EVENT_TYPE *EventNo, SEM_EVENT_TYPE *EventList)
```

Defined in

*System*SEMLibB.c

For use with

Table-based code

Description

The function is used to find events that can trigger transitions or derive action expressions from the current state. All events are found by continuous calls to this function. Because the function will inquire events on the basis of the internal current state configuration, an event deduction should not be running.

The function must be enabled by the Coder command line option `-seminquiry1` or the corresponding GUI option.

Parameters

`EventNo`	A pointer to store the inquired event number.
`EventList`	A pointer to an array that holds the event numbers to be inquired. `EventList` must be terminated with the definition `EVENT_TERMINATION_ID`, which indicates the end of the array. If the pointer is `NULL`, all events are inquired.

Return value

See:

SES_FOUND, page 633

SES_RANGE_ERR, page 633

SES_OKAY, page 633

Example

```
#define STRLEN 80

/* Used event definitions are found in System.h file. */
const SEM_EVENT_TYPE KeyTable[] =
{
  E_KEY_F1,
  E_KEY_F2,
  E_KEY_F3,
  E_KEY_F4,
  E_KEY_F5,
  E_KEY_F6,
  E_KEY_F7,
  E_KEY_F8,
  E_KEY_F9,
  E_KEY_F10,
  E_KEY_F11,
  E_KEY_F12,
  EVENT_TERMINATION_ID
};

/* Print active keys. */
unsigned char PrintActiveKeys (void)
{
  char Str[STRLEN];
  unsigned char CC = SES_OKAY;
  SEM_EVENT_TYPE EventNo = EVENT_UNDEFINED;

  if ((CC = SEM_Inquiry ()) == SES_OKAY)
  {
    printf ("\nActive event number:");
    while ((CC = SEM_GetInput (&EventNo, KeyTable)) == SES_FOUND)
    {
      /* To print the name of the event, use SEM_Name instead */
      if (SEM_Expl (EVENT_TYPE, EventNo, Str, STRLEN) ==
        SES_OKAY)
        printf ("\n%s = %d", Str, EventNo);
    }
  }
}
```

SEM_GetInputAll

Syntax

```
unsigned char SEM_GetInputAll
  (SEM_EVENT_TYPE *EventVector,
   SEM_EVENT_TYPE *EventList, SEM_EVENT_TYPE MaxSize)
```

Defined in *System*SEMLibB.c

For use with Table-based code

Description This function is used for finding all events that can trigger transitions. All events are found by one call to this function. Because the function will inquire events on the basis of the internal current state configuration, an event deduction should not be running.

The function must be enabled by the Coder command line option `-semgetinputall1` or the corresponding GUI option.

Parameters

`EventVector`	A pointer to an array in which to store the inquired events. The array is terminated with the definition `EVENT_TERMINATION_ID` on success.
`EventList`	A pointer to an array that holds the event numbers to be inquired. `EventList` must be terminated with the definition `EVENT_TERMINATION_ID`, which indicates the end of the array. If the pointer is `NULL`, all events are inquired.
`MaxSize`	The maximum length of the event vector including the definition `EVENT_TERMINATION_ID`.

Return value See:

SES_BUFFER_OVERFLOW, page 632

SES_RANGE_ERR, page 633

SES_OKAY, page 633

Example

```
#define STRLEN 80

/* Used event definitions are found in the System.h file. */
const SEM_EVENT_TYPE KeyTable[] =
{
  E_KEY_F1,
  E_KEY_F2,
  E_KEY_F3,
  E_KEY_F4,
  E_KEY_F5,
  E_KEY_F6,
  E_KEY_F7,
  E_KEY_F8,
  E_KEY_F9,
  E_KEY_F10,
  E_KEY_F11,
  E_KEY_F12,
  EVENT_TERMINATION_ID
};

/* Print active keys. */
unsigned char PrintActiveKeys (void)
{
  char Str[STRLEN];
  unsigned char CC = SES_OKAY;
  SEM_EVENT_TYPE EventList[13];
  int i;

  if ((CC = SEM_Inquiry ()) == SES_OKAY)
  {
    printf ("\nActive event number:");
    while ((CC = SEM_GetInput (&EventNo, KeyTable, 13)) ==
            SES_FOUND)
    {
      i = 0;
      while (EventList[i] != EVENT_TERMINATION_ID)
      {
        /* To print the name, call SEM_NameABS instead */
        if (SEM_ExplAbs (EVENT_TYPE, EventList[i], &Str) ==
            SES_OKAY)
            printf ("\n%s = %d", Str, EventList[i++]);
      }
    }
  }
}
```

SEM_Init

Syntax

```
void SEM_Init (void)
```

Defined in

*System*SEMLibB.c

For use with

Table-based code

Description

This function initializes the Visual State system and must be called before any other functions are called.

`SEM_Init` is called automatically by `SEM_InitAll`, which means that you should normally not need to call `SEM_Init`.

Parameters

None.

Return value

None.

Example

None.

SEM_InitAll

Syntax

```
#include "semlibb.h"
void SEM_InitAll (void)
```

Defined in

*System*SEMLibB.c

For use with

Table-based code

Description

This function wraps all initialization functions. The function calls the following functions in the listed order, provided that they exist:

```
SEM_Init
SEM_InitExternalVariables
SEM_InitInternalVariables
SEM_InitSignalQueue
SEM_InitInstances
```

The function must be enabled by the Coder command line option `-seminitall1` or the corresponding GUI option.

Parameters

None.

Return value

None.

Example	None.

SEM_InitExternalVariables

Syntax	`void SEM_InitExternalVariables (void)`
Defined in	*System*Data.c
For use with	Table-based code
Description	This function initializes the external variables in the system and must be called together with the `SEM_Init` function. The function is auto-generated by the Coder during the code generation of a system if any external variables are present, and the Coder option `-iew` has been set. `SEM_InitExternalVariables` is called automatically by `SEM_InitAll`.
Parameters	None.
Return value	None.
Example	None.

SEM_InitInstances

Syntax	`unsigned char SEM_InitInstances (void)`
Defined in	*System*SEMLibB.c
For use with	Table-based code
Description	This function initializes a number of instances of a system. The instance is handled in pseudo-parallel using the `SEM_SetInstance` function. The actual number of instances is determined by the information in the system. `SEM_InitInstances` is called automatically by `SEM_InitAll`.
Parameters	None.

Return value

See:

SES_OKAY, page 633

Example

```
unsigned char Instance (SEM_EVENT_TYPE EventNo,
  SEM_INSTANCE_TYPE InstanceNo)
{
  /* Declare action expression variable. */
  SEM_ACTION_EXPRESSION_TYPE ActionExpress;

  /* Set active instance. */
  if (SEM_SetInstance (InstanceNo) != SES_OKAY)
    return (FALSE);

  if (VSDeduct (EventNo) != SES_OKAY)
    return (FALSE);

  return (TRUE)
}


void Task (void)
{
  SEM_INSTANCE_TYPE InstanceNo = 0;

  /*
   * Declare and initialize. In this case the
   * reset event is SE_RESET.
   */
  SEM_EVENT_TYPE EventNo = SE_RESET;

  /* Initialize the System and related data. */
  SEM_InitAll ();
  for (InstanceNo = 0; InstanceNo < VS_NOF_INSTANCES;
    InstanceNo++)
  {
    Instance (EventNo, InstanceNo);
  }
```

```
  /* Do forever. */
  while (1)
  {
    /*
     * Get new event and map it to VS System events and
     * instance.
     */
    MapEvent (&EventNo, &InstanceNo);
    /* Process the event. */
    if (Instance (EventNo, InstanceNo) != TRUE)
      ErrorHandling ();
  }
}
```

See also

The Visual State system, page 123.

SEM_InitInternalVariables

Syntax

`void SEM_InitInternalVariables (void)`

Defined in

`System``Data.c`

For use with

Table-based code

Description

This function initializes the internal variables in the system and must be called together with the `SEM_Init` function.

The function is auto-generated by the Coder during the code generation of a system if any internal variables are present, and the Coder option `-iev` has been set to 1.

`SEM_InitInternalVariables` is called automatically by `SEM_InitAll`.

Parameters

None.

Return value

None.

Example

None.

SEM_InitSignalQueue

Syntax

`void SEM_InitSignalQueue (void)`

Defined in

`System``SEMLibB.c`

For use with — Table-based code

Description — This function initializes the signal queue in a Visual State system and must be called together with the `SEM_Init` function. The function will only be available if the signal queue is enabled and the system contains signals.

`SEM_InitSignalQueue` is called automatically by `SEM_InitAll`.

Parameters — None.

Return value — None.

Example — None.

SEM_Inquiry

Syntax — `unsigned char SEM_Inquiry (void)`

Defined in — *System*`SEMLibB.c`

For use with — Table-based code

Description — This function prepares for finding events that can trigger changes in the current state. All events are found by continuous calls to the function `SEM_GetInput` or one call to `SEM_GetInputAll`.

As the function will inquire events on the basis of the internal current state configuration, `SEM_Inquiry` can only be used if the previously called function is `SEM_Init`.

The function must be enabled by the Coder command line option `-seminquiry1` or the corresponding GUI option.

Parameters — None.

Return value — See:

SES_ACTIVE, page 632

SES_OKAY, page 633

Example

```
#define STRLEN 80

/* Print active events */
unsigned char PrintActiveEvents (void)
{
  char Str[STRLEN];
  unsigned char CC = SES_OKAY;
  SEM_EVENT_TYPE EventNo = EVENT_UNDEFINED;

  if ((CC = SEM_Inquiry ()) == SES_OKAY)
  {
    printf ("\nActive event numbers:");
    while ((CC = SEM_GetInput (&EventNo, NULL)) == SES_FOUND)
    {
      if (SEM_Name (EVENT_TYPE, EventNo, Str, STRLEN)
          == SES_OKAY)
        printf ("\n%s = %d", Str, EventNo);
    }
  }
  return (CC);
}
```

SEM_Machine

Syntax

```
unsigned char SEM_Machine (SEM_STATE_TYPE StateNo,
                          SEM_STATE_MACHINE_TYPE *StateMachineNo)
```

Defined in

*System*SEMLibB.c

For use with

Table-based code

Description

This function returns the state machine index of the specified state.

The function must be enabled by the Coder command line option `-semmachine1` or the corresponding GUI option.

Parameters

`StateNo`	Contains the state index number.
`StateMachineNo`	Contains a pointer for storing the state machine index number found of the specified state.

Return value

See:

SES_RANGE_ERR, page 633

SES_FOUND, page 633

Example

```
#include "SystemSEMLibB.h"
/*
  * The function is used for turning on/off a standby LED
  */
unsigned char CheckStandby (void)
{
  unsigned char CC;
  SEM_STATE_TYPE StateNo;
  SEM_STATE_MACHINE_TYPE StateMachine;

  /* State STATE_STANDBY defined in System.h file. */
  if ((CC = SEM_Machine (STATE_STANDBY, &StateMachine)) ==
    SES_FOUND)
  {
    if ((CC = SEM_State (StateMachine, &StateNo)) == SES_FOUND)
    {
      if (StateNo == STATE_STANDBY)
        StandbyLED = TRUE;
      else
        StandbyLED = FALSE;
    }
  }
  return (CC);
}
```

See also

-*snm*, page 737.

SEM_Name

Syntax

```
unsigned char SEM_Name (unsigned char IdentType,
                        SEM_EXPLANATION_TYPE IdentNo,
                        char *Text,
                        unsigned short MaxSize)
```

Defined in

*System*SEMLibB.c

For use with

Table-based code

Description

This deprecated function is provided for backward compatibility and should not be used. Instead use the function VSElementName, see *VSElementName*, page 617.

This function gets the ASCII name of the specified identifier and can only be used (and compiled) when at least one type of name is included in the system.

The function must be enabled in combination with enabling generation of the names you want to get. For example, set `-semname1` and `-txte1`. This will enable the function, and enable generating names for events. These Coder options can also be enabled by setting the corresponding GUI options.

Parameters

`IdentType`	Must contain the type of the identifier, `EVENT_TYPE` or `STATE_TYPE`. Note that `ACTION_TYPE` is deprecated because there is no advantage to using it here.
`IdentNo`	Must contain the index number of an identifier.
`Text`	Must contain a pointer to a text string. If the function terminates successfully, the text string contains the name of the specified identifier.
`MaxSize`	Specifies the maximum length of the text, including the `NULL` termination character.

Return value

See:

SES_RANGE_ERR, page 633

SES_TEXT_TOO_LONG, page 634

SES_TYPE_ERR, page 634

SES_OKAY, page 633

Example

See *SEM_Inquiry*, page 604.

SEM_NameAbs

Syntax

```
unsigned char SEM_NameAbs (unsigned char IdentType,
                           SEM_EXPLANATION_TYPE IdentNo,
                           char **Text)
```

Defined in

System`SEMLibB.c`

For use with

Table-based code

Description	This function gets a pointer to the ASCII name of the specified identifier. The function must be enabled in combination with enabling generation of the names you want to get. For example, set `-semnameabs1` and `-txte1`. This will enable the function, and enable generating names for events. These Coder options can also be enabled by setting the corresponding GUI options.

Parameters

`IdentType`	Must contain the type of the identifier which can be `EVENT_TYPE` or `STATE_TYPE`. Note that `ACTION_TYPE` is deprecated because there is no advantage using it here.
`IdentNo`	Must contain the index number of an identifier.
`Text`	Must contain an address of a pointer to a text string. If the function terminates successfully, the text pointer contains the address of the name of the specified identifier.

Return value

See:

SES_OKAY, page 633

SES_RANGE_ERR, page 633

SES_TYPE_ERR, page 634

Example

See *SEM_GetInputAll*, page 597.

SEM_SetInstance

Syntax	`unsigned char SEM_SetInstance (SEM_INSTANCE_TYPE Instance)`
Defined in	*`System`*`SEMLibB.c`
For use with	Table-based code
Description	This function is used for setting the currently active instance of the system. The instance remains active until the next call to this function. The function must only be called between completed macrosteps, not in the middle of a macrostep. For example, do not call the function directly after a call to `VSDeduct`.

Parameters

`Instance`	The instance to be handled.

Return value

See:

SES_ACTIVE, page 632

SES_RANGE_ERR, page 633

SES_OKAY, page 633

Example

See *SEM_InitInstances*, page 601.

SEM_SignalQueueInfo

Syntax

```
void SEM_SignalQueueInfo (SEM_SIGNAL_QUEUE_TYPE *NofSignals)
```

Defined in

*System*SEMLibB.c

For use with

Table-based code

Description

This function returns information about the signal queue. The function will only be available if the signal queue is enabled and the Visual State system contains signals.

The function must be enabled by the Coder command line option `-semsignalqueueinfo1` or the corresponding GUI option.

Parameters

NofSignals	Number of signals in the signal queue.

Return value

None.

Example

None.

SEM_State

Syntax

```
unsigned char SEM_State (SEM_STATE_MACHINE_TYPE StateMachineNo,
                         SEM_STATE_TYPE *StateNo)
```

Defined in

*System*SEMLibB.c

For use with

Table-based code

Description

This function returns the current state of the specified state machine.

The function must be enabled by the Coder command line option `-semstate1` or the corresponding GUI option.

Parameters

`StateMachineNo`	Contains the state machine number.
`StateNo`	Contains a pointer for storing the current state of the specified state machine.

Return value

See:

SES_FOUND, page 633

SES_RANGE_ERR, page 633

Example

```
void Task (void)
{
  SEM_STATE_TYPE StateNo = STATE_UNDEFINED;
  SEM_STATE_MACHINE_TYPE i;
  SEM_ACTION_EXPRESSION_TYPE actionExpressNo;
  unsigned char cc;

  /*
   * Declare and initialize event variable.
   * In this case the reset event is SE_RESET.
   */
  SEM_EVENT_TYPE EventNo = SE_RESET;

  /* Initialize the VS System. */
  SEM_InitAll ();

  /* Do forever. */
  while (1)
  {
    if ((cc = VSDeduct(EventNo)) != SES_OKAY)
      ErrorHandling ();
```

```
    for (i = 0; i < VS_NOF_STATE_MACHINES; i++)
    {
      if (SEM_State (i, &StateNo) != SES_FOUND)
        printf ("\nState machine %d is in undefined state", i);
      else
        /* Print state machine number and state number. */
        printf ("\nState machine %d: state %d", i, StateNo);
    }

    /* Get new event and map it to VS System events. */
    MapEvent (&EventNo);
  }
}
```

See also

-*snm*, page 737.

SEM_StateAll

Syntax

```
unsigned char SEM_StateAll
              (SEM_STATE_TYPE *StateVector,
               SEM_STATE_MACHINE_TYPE MaxSize)
```

Defined in

System`SEMLibB.c`

For use with

Table-based code

Description

This function returns the active state of all state machines.

The function must be enabled by the Coder command line option `-semstateall1` or the corresponding GUI option.

Parameters

`StateVector`	A pointer to an array in which to store the current state configuration.
`MaxSize`	Specifies the length of the destination array. Must be equal to or longer than the number of state machines.

Return value

See:

SES_BUFFER_OVERFLOW, page 632

SES_FOUND, page 633

Example

```
void Task (void)
{
  SEM_STATE_TYPE StateList[VS_NOF_STATE_MACHINES];
  SEM_STATE_MACHINE_TYPE i;
  SEM_ACTION_EXPRESSION_TYPE actionExpressNo;
  unsigned char cc;

  /*
   * Declare and initialize. In this case the
   * reset event is SE_RESET.
   */
  SEM_EVENT_TYPE EventNo = SE_RESET;

  /* Initialize the VS System. */
  SEM_InitAll ();
  /* Do forever. */
  while (1)
  {
    if ((cc = VSDeduct(EventNo)) != SES_OKAY)
      ErrorHandling ();

    if (SEM_StateAll (StateList, VS_NOF_STATE_MACHINES) !=
                      SES_FOUND)
      printf ("\nCannot access states.");
    else
    {
      /* Print state machine number and state number. */
      for (i = 0; i < VS_NOF_STATE_MACHINES; i++)
        printf ("\nState machine %d: state %d", i,
          StateList[i]);
    }

    /* Get new event and map it to VS System events. */
    MapEvent (&EventNo);
  }
}
```

VSDeduct

Syntax `VS_UINT8 VSDeduct(SEM_EVENT_TYPE EventNo, ...);`

Defined in *System*`SEMLibB.c`

For use with Readable code, or if you enabled the `-vsdeduct` Coder option.

Description

This function deduces all the relevant action expressions on the basis of the given event, the internal current state configuration and the transitions in the Visual State system. All the relevant action expressions are then called and all the next states are changed.

Parameters

`EventNo`	The event number to be processed. If at least one event has parameters, the function call must include one argument for each parameter declared in the parameter list for each event.

Return value

See:

SES_CONTRADICTION, page 632

SES_FOUND, page 633

SES_OKAY, page 633

SES_RANGE_ERR, page 633

SES_SIGNAL_QUEUE_FULL, page 634

Example

```
/*
 * Event E_Event1 without parameters
 */
if (VSDeduct (E_Event1) != SES_OKAY)
  ErrorHandling ();

/*
 * Event E_Event2 with two parameters:
 * Argument 1: unsigned int Par1
 * Argument 2: unsigned short Par2
 */
if (VSDeduct (E_Event2, Par1, Par2) != SES_OKAY)
  ErrorHandling ();

void Task (void)
{
  unsigned char cc;
  SEM_EVENT_TYPE eventNo = SE_RESET;

  /* Initialize the VS System. */
  VSInitAll();
```

```
  /* do forever */
  while (1)
  {
   cc = VSDeduct(eventNo);
     /*
      * If you enabled the semnextstatechg Coder option
      * if (cc == SES_FOUND)
      * {
      *    /* react to a change in some state */
      * }
      */
    if (cc != SES_OKAY && cc != SES_FOUND)
      handleError(cc);
    /* Get new event and map it to VS system events */
    MapEvent (&eventNo);
  }
}
```

VSDeductInstance

Syntax

```
VS_UINT8 VSDeductInstance(VS_UINT16 instance,
                          SEM_EVENT_TYPE EventNo, ...);
```

Parameters

`instance`	The instance to work on in the Visual State system.
`EventNo`	The event number to be processed. If at least one event has parameters, the function call must include one argument for each parameter declared in the parameter list for each event.

Return value

See:

SES_CONTRADICTION, page 632

SES_FOUND, page 633

SES_OKAY, page 633

SES_RANGE_ERR, page 633

SES_SIGNAL_QUEUE_FULL, page 634

Defined in

*System*SEMLibB.c

For use with

Readable code

Description

This function deduces all the relevant action expressions on the basis of the given event, the internal current state configuration, and the transitions in the Visual State system for the given instance. All the relevant action expressions are then called and all the following states are changed.

Example

```
/*
 * Event E_Event1 without parameters
 */
if (VSDeduct (instance, E_Event1) != SES_OKAY)
  ErrorHandling ();

/*
 * Event E_Event2 with two parameters:
 * Argument 1: unsigned int Par1
 * Argument 2: unsigned short Par2
 */
if (VSDeduct (instance, E_Event2, Par1, Par2) != SES_OKAY)
  ErrorHandling ();

void Task (void)
{
  unsigned char cc;
  /*
   * You need to keep track of which instance you work with.
   * Here it is just set to 0 as an example.
   */
  VS_UINT16 instance = 0;
  SEM_EVENT_TYPE eventNo = SE_RESET;

  /* Initialize the VS System. */
  VSInitAll();

  /* do forever */
  while (1)
  {
    cc = VSDeductInstance(instance, eventNo);
    /*
     * If you enabled the -semnextstatechg Coder option
     * if (cc == SES_FOUND)
     * {
     *   /* react to a change in some state */
     * }
     */
    if (cc != SES_OKAY && cc != SES_FOUND)
      handleError(cc);
```

```
        /* Get new event and instance and map it to system events */
        MapEvent (&eventNo, &instance);
      }
    }
```

VSElementExpl

Syntax

```
VSResult VSElementExpl(VS_UINT8 IdentType,
                       SEM_EXPLANATION_TYPE IdentNo,
                       char const **Text);
```

Parameters

`IdentType`	Must contain one of the identifier types, `EVENT_TYPE` or `STATE_TYPE`.
`IdentNo`	Must contain the index number of an identifier.
`Text`	Must contain an address of a pointer to a text string. If the function terminates successfully, the text pointer contains the address of the explanation of the specified identifier.

Return value

See:

SES_OKAY, page 633

SES_RANGE_ERR, page 633

SES_TYPE_ERR, page 634

Defined in

System`SEMLibB.c`

For use with

Readable code, or all types of generated code if the Coder options `-maximummisra1` and `-vselementexpl1` have been set.

Description

This function gets a pointer to the explanation of the specified identifier.

You must also enable generation of the explanations you want to get. For example, set the Coder options `-vselementexpl1` and `-txte2`. This will enable the function, and enable generating explanations for events.

Example

```
void dumpEvent (SEM_EXPLANATION_TYPE eventNo)
{
  char const *expl;
  unsigned char cc;
  if ((cc = VSElementExpl((unsigned char)EVENT_TYPE,
                           eventNo,
                           &name)) != SES_OKAY)
  {
    /*
     * Handle the error by reporting it to the environment.
     * You probably need to enable the texts in the
     * Coder options.
     */
    return;
  }
  printf("Event '%s' sent to the system.", expl);
}
```

VSElementName

Syntax

```
VS_UINT8 VSElementName(VS_UINT8 IdentType,
                       SEM_EXPLANATION_TYPE IdentNo,
                       char const **Text);
```

Parameters

`IdentType`	Must contain the type of the identifier which can be `EVENT_TYPE` or `STATE_TYPE`.
`IdentNo`	Must contain the index number of an identifier.
`Text`	Must contain an address of a pointer to a text string. If the function terminates successfully, the text pointer contains the address of the name of the specified identifier.

Return value

See:

SES_OKAY, page 633

SES_RANGE_ERR, page 633

SES_TYPE_ERR, page 634

Defined in

`*System*SEMLibB.c`

For use with

All types of readable code if you enabled the `-vselementname` Coder option.

Description

This function gets a pointer to the ASCII name of the specified identifier.

You must also enable generation of the names you want to get. For example, set the Coder options `-semnameabs1` and `-txte1`. This will enable the function, and enable generating names for events.

Example

See *SEM_Inquiry*, page 604.

VSForceState

Syntax

`VS_UINT8 VSForceState(SEM_STATE_TYPE StateNo);`

Parameters

`StateNo`	Contains the state index number.

Return value

See:

SES_OKAY, page 633

SES_RANGE_ERR, page 633

Defined in

`*System*SEMLibB.c`

For use with

Readable code

Description

This function is used for forcing the internal state configuration into the specified state. This is useful if you want to reestablish the internal state configuration after a power failure of the target system. Before calling this function the first time after a power failure, the `VSInitAll` function should be called to initialize the other internal variables of the system.

The state configuration established by calling `VSForceState` must have been stored in EEPROM before the power failure.

Note: This function should be used with caution. The internal state configuration could be forced to a configuration that is not reachable by executing the model itself.

The function must be enabled by the Coder option `-semforcestate1`.

Example

```
/*
 * This function should only be called after a power failure
 * to reestablish the internal state variables.
 */
void PowerUp (void)
{
  SEM_STATE_TYPE StateNo;
  SEM_STATE_MACHINE_TYPE i;
  /* Initialize the Visual State system. */
  VSInitAll ();
  for (i = 0; i < VS_NOF_STATE_MACHINES; i++)
  {
    /* Get state configuration from EEPROM. */
    EEPROMState (i, &StateNo);
    if (VSForceState (StateNo) != SES_OKAY)
      ErrorHandling ();
  }
}
```

VSForceStateInstance

Syntax

```
VS_UINT8 VSForceStateInstance(VS_UINT16 instance,
                              SEM_STATE_TYPE StateNo);
```

Parameters

`instance`	The instance of the system to change.
`StateNo`	Contains the state index number.

Return value

See:

SES_OKAY, page 633

SES_RANGE_ERR, page 633

Defined in

`*System*SEMLibB.c`

For use with

Readable code

Description

This function is used for forcing the internal state configuration into the specified state for the given instance of a system. This is useful if you want to reestablish the internal state configuration after a power failure of the target system. Before calling this function the first time after a power failure, the `VSInitAll` function should be called to initialize the other internal variables of the system.

The state configuration established by calling `VSForceState` must have been stored in EEPROM before the power failure.

Note: This function should be used with caution. The internal state configuration could be forced to a configuration that is not reachable by executing the model itself.

The function must be enabled by the Coder option `-semforcestate1`.

Example

```
/*
 * This function should only be called after a power failure
 * to reestablish the internal state variables.
 */
void PowerUp (void)
{
  SEM_STATE_TYPE StateNo;
  SEM_STATE_MACHINE_TYPE i;
  VS_UINT16 instance;
  /* Initialize the Visual State system. */
  VSInitAll ();
  for (instance = 0; instance < VS_NOF_INSTANCES; instance++)
  {
    for (i = 0; i < VS_NOF_STATE_MACHINES; i++)
    {
      /* Get state configuration from EEPROM. */
      EEPROMState (i, instance, &StateNo);
      if (VSForceStateInstance (instance, StateNo) != SES_OKAY)
        ErrorHandling ();
    }
  }
}
```

*System*VSGetCurrentStateTree

Syntax

```
VSResult SystemVSGetCurrentStateTree (char * buf, size_t const
                                      bufSize)
```

Defined in

System.c

Description

This function copies the strings that represent the current state tree into the buffer. Each entry ends with a semicolon.

Argument

`buf`	A pointer to a buffer.
`bufSize`	The size of the buffer `buf`.

Return value — See:

SES_OKAY, page 633

SES_TEXT_TOO_LONG, page 634

Example — None.

*System*VSGetMaxCurrentStateTree

Syntax — `size_t SystemVSGetMaxCurrentStateTree (void)`

Defined in — `System.c`

Description — This function returns the required size of the `VSGetCurrentStateTree` buffer.

Argument — None.

Return value — The required size of the `VSGetCurrentStateTree` buffer.

Example — None.

VSInitAll

Syntax — `void VSInitAll(void)`

Parameters — None.

Return value — None.

Defined in — `SystemSEMLibB.c`

For use with — Readable code, or if you enabled the `-vsinitall` Coder option.

Description — This function wraps all initialization functions. The function calls the following functions in the listed order, provided that they exist:

- `VSInit`
- `VSInitExternalVariables`
- `VSInitInternalVariables`
- `VSInitSignalQueue`

- `VSInitInstances`

The function must be enabled by the Coder option `-seminitall1` or `-vsinitall1`.

Example — See *VSDeduct*, page 612.

VSInitExternalVariables

Syntax — `void VSInitExternalVariables(void)`

Parameters — None.

Return value — None.

Defined in — *System*`SEMLibB.c`

For use with — Readable code

Description — This function initializes the external variables in the system and must be called together with the `VSInitAll` function.

The function is automatically generated by the Coder during the code generation of a system if any external variables are present, and the Coder option `-iev` has been set.

`VSInitExternalVariables` is called automatically by `VSInitAll`.

Example — None.

VSInitInternalVariables

Syntax — `void VSInitInternalVariables(void)`

Parameters — None.

Return value — None.

Defined in — *System*`SEMLibB.c`

For use with — Readable code

Description — This function initializes the internal variables in the system and must be called together with the `VSInitAll` function.

The function is automatically generated by the Coder during the code generation of a system if any internal variables are present, and the Coder option `-iiv` has been set.

`VSInitInternalVariables` is called automatically by `VSInitAll`.

Example

None.

VSInquiry

Syntax

```
VS_UINT8 VSInquiry(SEM_EVENT_TYPE* FoundEvents,
                   VS_UINT Size,
                   SEM_EVENT_TYPE* EventList);
```

Parameters

`FoundEvents`	Array to fill with found events. The array will be terminated with `EVENT_UNDEFINED`.
`Size`	The size of the array to fill with active events.
`EventList`	Pointer to an array that holds the event numbers that can be inquired. `EventList` must be terminated with `EVENT_UNDEFINED`. If the pointer is `NULL`, then all events can be inquired.

Return value

See:

SES_BUFFER_OVERFLOW, page 632

SES_OKAY, page 633

SES_RANGE_ERR, page 633

Defined in

`System``SEMLibB.c`

For use with

Readable code

Description

The function is used for finding events that can trigger transitions or derive action expressions from the current state. All events are found by a call to this function. Because the function will inquire events on the basis of the internal current state configuration, an event deduction should not be running.

The function must be enabled by the Coder option `-seminquiry1`.

Example

```
/* #include SystemSEMLibB.h before this. */

unsigned char PrintActiveKeys (void)
{
  /* declare big enough to hold all - so we can use it below */
  enum {size = VS_NOF_EVENTS + 1};
  SEM_EVENT_TYPE events[size] =
  {
    E1,
    E2,
    E3,
    EVENT_UNDEFINED
  };
  unsigned char cc;
  unsigned count = 0;

  /* to inquire the 3 events specified in the list
     above (E1, E2, E3) */
  if ((cc = VSInquiry(events, size, events)) != SES_OKAY)
    handleError("VSInquiry", cc);
  /*
   * if you want to inquire all active events use this line:
   * if ((cc = VSInquiry(events, size, NULL)) != SES_OKAY)
   */
  while (events[count] != EVENT_UNDEFINED)
  {
    char const *pName;
    if ((cc = VSElementName(EVENT_TYPE,
                events[count], &pName)) != SES_OKAY)
      handleError("VSElementName", cc);
    /* to print the explanation call VSElementExpl to get the
       explanation instead of the name */
    printf("Found active event: %s", pName);
    ++count;
  }
}
```

VSInquiryInstance

Syntax

```
VS_UINT8 VSInquiryInstance(VS_UINT16 instance,
                           SEM_EVENT_TYPE* FoundEvents,
                           VS_UINT Size,
                           SEM_EVENT_TYPE* EventList);
```

Parameters

`instance`	The instance of the system to inquire the active events for.

`FoundEvents`	Array to fill with found events. The array will be terminated with `EVENT_UNDEFINED`.
`Size`	The size of the array to fill with active events.
`EventList`	Pointer to an array that holds the event numbers that can be inquired. `EventList` must be terminated with `EVENT_UNDEFINED`. If the pointer is `NULL`, then all events can be inquired.

Return value

See:

SES_BUFFER_OVERFLOW, page 632

SES_OKAY, page 633

SES_RANGE_ERR, page 633

Defined in

`System``SEMLibB.c`

For use with

Readable code

Description

This function is used to find events that can trigger transitions or derive action expressions from the current state for the given instance. All events are found by a call to this function. Because the function will inquire events on the basis of the internal current state configuration, an event deduction should not be running.

The function must be enabled by the Coder option `-seminquiry1`.

Example

```
/* #include SystemSEMLibB.h before this. */

unsigned char PrintActiveKeys (VS_UINT16 instance)
{
  /* declare big enough to hold all - so we can use it below */
  enum {size = VS_NOF_EVENTS + 1};
  SEM_EVENT_TYPE events[size] =
  {
    E1,
    E2,
    E3,
    EVENT_UNDEFINED
  };
  unsigned char cc;
  unsigned count = 0;
```

```
  /* to query the 3 events specified in the list
     above (E1, E2, E3) */
  if ((cc = VSInquiryInstance(instance, events, size, events))
       != SES_OKAY)
    handleError("VSInquiry", cc);
  /*
   * if you want to query all active events use this line:
   * if ((cc = VSInquiryInstance(instance, events, size, NULL))
   *       != SES_OKAY)
   */
  while (events[count] != EVENT_UNDEFINED)
  {
    char const *pName;
    if ((cc = VSElementName(EVENT_TYPE,
              events[count], &pName)) != SES_OKAY)
      handleError("VSElementName", cc);
    /* to print the explanation call VSElementExpl to get the
       explanation instead of the name */
    printf("Found active event: %s", pName);
    ++count;
  }
}
```

VSMachine

Syntax

```
VS_UINT8 VSMachine(SEM_STATE_TYPE StateNo,
                   SEM_STATE_MACHINE_TYPE *StateMachineNo);
```

Parameters

`StateNo`	Contains the state index number.
`StateMachineNo`	Contains a pointer for storing the state machine index number of the specified state.

Return value

See:

SES_FOUND, page 633

SES_RANGE_ERR, page 633

Defined in

System`SEMLibB.c`

For use with

Readable code

Description

This function returns the state machine index of the specified state.

The function must be enabled by the Coder option `-semmachine1`.

Example

```
#include "SystemSEMLibB.h"
/*
 * The function is used for turning on/off a standby LED. It must
 * be called, just after VSDeduct has been called.
 */
unsigned char CheckStandby (void)
/* or: unsigned char CheckStandby (VS_UINT16 instance)
{
  unsigned char CC;
  SEM_STATE_TYPE state;
  SEM_STATE_MACHINE_TYPE mach;
  if ((CC = VSMachine (STATE_STANDBY, &mach)) == SES_FOUND)
  {
    if ((CC = VSState (mach, &state)) == SES_FOUND)
    /* or: if ((CC = VSStateInstance (instance, mach, &state))
                == SES_FOUND) */
    {
      if (state == STATE_STANDBY)
        StandbyLED = TRUE;
      else
        StandbyLED = FALSE;
    }
  }
  return CC;
}
```

VSState

Syntax

```
VS_UINT8 VSState(SEM_STATE_MACHINE_TYPE StateMachineNo,
                 SEM_STATE_TYPE *StateNo);
```

Parameters

`StateMachineNo`	Contains the state machine number.
`StateNo`	Contains a pointer for storing the current state of the specified state machine.

Return value

See:

SES_FOUND, page 633

SES_RANGE_ERR, page 633

Defined in

`System`SEMLibB.c

For use with

Readable code

Description

This function returns the current state of the specified state machine.

The function must be enabled by the Coder option `-semstate1`.

Example

See *VSMachine*, page 626.

VSStateAll

Syntax

```
VS_UINT8 VSStateAll(SEM_STATE_TYPE *StateVector,
                    SEM_STATE_MACHINE_TYPE MaxSize);
```

Parameters

`StateVector`	A pointer to an array in which to store the current state configuration.
`MaxSize`	Specifies the length of the destination array. Must be equal to or longer than the number of state machines.

Return value

See:

SES_BUFFER_OVERFLOW, page 632

SES_FOUND, page 633

Defined in

`System`SEMLibB.c

For use with

Readable code

Description

This function returns the active state of all state machines.

The function must be enabled by the Coder option `-semstateall1`.

Example

```
void Task (void)
{
  SEM_STATE_TYPE StateList[VS_NOF_STATE_MACHINES];
  SEM_STATE_MACHINE_TYPE i;
  unsigned char cc;
  SEM_EVENT_TYPE EventNo = SE_RESET;

  VSInitAll ();
  /* Do forever. */
  while (1)
  {
    if ((cc = VSDeduct(EventNo)) != SES_OKAY)
      ErrorHandling (cc);
    if (VSStateAll (StateList, VS_NOF_STATE_MACHINES)
                    != SES_FOUND)
      printf ("\nCannot access states.");
    else
    {
      /* Print state machine number and state number. */
      for (i = 0; i < VS_NOF_STATE_MACHINES; i++)
        printf ("\nState machine %d: state %d", i, StateList[i]);
    }
    /* Get new event and map it to VS System events. */
    MapEvent (&EventNo);
  }
}
```

VSStateAllInstance

Syntax

```
VS_UINT8 VSStateAllInstance(VS_UINT16 instance,
                            SEM_STATE_TYPE *StateVector,
                            SEM_STATE_MACHINE_TYPE MaxSize);
```

Parameters

`instance`	The instance number to work with.
`StateVector`	A pointer to an array in which to store the current state configuration.
`MaxSize`	Specifies the length of the destination array. Must be equal to or longer than the number of state machines.

Return value

See:

SES_BUFFER_OVERFLOW, page 632

SES_FOUND, page 633

Defined in

System`SEMLibB.c`

For use with

Readable code

Description

This function returns the active state of all state machines for the given instance of the system.

The function must be enabled by the Coder option `-semstateall1`.

Example

```
void Task (void)
{
  SEM_STATE_TYPE StateList[VS_NOF_STATE_MACHINES];
  SEM_STATE_MACHINE_TYPE i;
  unsigned char cc;
  SEM_EVENT_TYPE EventNo = SE_RESET;

  VSInitAll ();
  /* start by sending the reset event to all instances */
  for (instance = 0; instance < VS_NOF_INSTANCES; instance++)
  {
    if ((cc = VSDeductInstance(instance, EventNo)) != SES_OKAY)
      ErrorHandling (cc);
  }
  /* Get new event and instance and map it to system events. */
  MapEvent (&EventNo, &instance);

  /* Do forever. */
  while (1)
  {
    if ((cc = VSDeductInstance(instance, EventNo)) != SES_OKAY)
      ErrorHandling (cc);
    if (VSStateAllInstance (instance, StateList,
        VS_NOF_STATE_MACHINES) != SES_FOUND)
      printf ("\nCannot access states for the instance %d.",
              instance);
```

```
    else
    {
      /* Print state machine number and state number. */
      for (i = 0; i < VS_NOF_STATE_MACHINES; i++)
        printf ("\nInstance %d, State machine %d: state %d",
                instance, i, StateList[i]);
    }
    /* Get new event and map it to VS System events. */
    MapEvent (&EventNo, &instance);
  }
}
```

VSStateInstance

Syntax

```
VS_UINT8 VSStateInstance(VS_UINT16 instance,
                         SEM_STATE_MACHINE_TYPE StateMachineNo,
                         SEM_STATE_TYPE *StateNo);
```

Parameters

`instance`	The instance number to work on.
`StateMachineNo`	Contains the state machine number.
`StateNo`	Contains a pointer for storing the current state of the specified state machine.

Return value

See:

SES_FOUND, page 633

SES_RANGE_ERR, page 633

Defined in

`*System*SEMLibB.c`

For use with

Readable code

Description

This function returns the current state of the specified state machine for the specified instance.

The function must be enabled by the Coder option `-semstate1`.

Example

See *VSMachine*, page 626.

Adaptive API return codes

The following pages give detailed reference information about each Adaptive API return code.

SES_ACTIVE

Return code `SES_ACTIVE`

Description The return code covers one of the following:

- An event deduction is started while an event inquiry is active. All inquired events have not been returned by the function `SEM_GetInput`.
- An event inquiry is started while an event deduction is active. All deduced action expressions have not been returned by the function `SEM_GetOutput`.

Solution The return code is a warning and maybe the application must be rewritten. An event inquiry and an event deduction should not be active at the same time.

SES_BUFFER_OVERFLOW

Return code `SES_BUFFER_OVERFLOW`

Description A destination buffer cannot hold the number of items found.

Solution Call the function with an extended buffer as the destination.

SES_CONTRADICTION

Return code `SES_CONTRADICTION`

Description A contradiction has been detected between two states in a state machine.

Solution Check the system with the Validator or the Verificator. You might need to change your model to avoid the conflict. The system should not contain any contradictions at runtime because that will cause the model to behave incorrectly and non-deterministically.

SES_EMPTY

Return code SES_EMPTY

Description No events have been given to the VSDeduct function before calling this function.

Solution Call the VSDeduct function with an event number.

SES_FOUND

Return code SES_FOUND

Description The called function has returned an identifier index number.

Solution If the function SEM_GetInput was called, the function can be called again to find more events or action expressions.

SES_NOT_INITIALIZED

Return code SES_NOT_INITIALIZED

Description The system has not been initialized.

Solution Call the initialization function for the system.

SES_OKAY

Return code SES_OKAY

Description Function performed successfully.

Solution Not applicable.

SES_RANGE_ERR

Return code SES_RANGE_ERR

Description A reference is being made to an identifier that does not exist. Note that the first index number is 0. If the system has 4 identifiers of the same type, and a function is called with a parameter value equal to 4, the function will return an SES_RANGE_ERR error. In this case the highest permitted variable value is 3.

Solution — Call with an index that is within the permitted range.

SES_SIGNAL_QUEUE_FULL

Return code — SES_SIGNAL_QUEUE_FULL

Description — The signal queue is full.

Solution — Increase the maximum signal queue size in your system.

SES_TEXT_TOO_LONG

Return code — SES_TEXT_TOO_LONG

Description — The requested text is longer than the specified maximum length.

Solution — Increase the maximum length.

SES_TYPE_ERR

Return code — SES_TYPE_ERR

Description — A text function has been called with the wrong identifier type, or the specified text is not included in the Visual State system.

Solution — Use the identifier type symbols (EVENT_TYPE, STATE_TYPE, or ACTION_TYPE) defined in the SEMLibB.h file. Set the Coder options so that the text is included in the generated code for the system.

Uniform API reference information

- Uniform API source files
- Summary of the Uniform API functions
- Descriptions of the Uniform API functions
- Uniform API return codes

Uniform API source files

The Uniform API will be generated in two files: *project*.c and *project*.h, where *project* reflects the name of your project. Most of the functions in the Uniform API have SMP as prefix and they take a pointer to a system context as a parameter to determine the system to operate on. This means that the API can operate on projects that contain multiple systems.

Unless otherwise stated, portability of the Uniform API is Standard C compliant.

Read more about:

- *Coder-generated source files for the Uniform API*, page 635

CODER-GENERATED SOURCE FILES FOR THE UNIFORM API

During the code generation phase, these sets of files are generated:

- Project-specific files
- System-specific files

These are the project-specific files:

project.h	Contains the declarations of the SMP functions; *project* reflects the name of your project.
project.c	Contains the implementations of the SMP functions; *project* reflects the name of your project.

These are the system-specific files (for each system):

source.c	Contains the core model logic of the system (primarily transitions).
source.h	Header files for *source*.c.

Summary of the Uniform API functions

This table summarizes the Uniform API functions:

Uniform API function	Description
*System*SEM_InitExternalVariables	Initializes the external variables in the system.
*System*SEM_InitInternalVariables	Initializes the internal variables in the system.
*Project*SEM_InitPrjExternalVariables	Initializes the external variables in the project.
SMP_Action	A macro that uses the VSAction function pointer table to call an action expression function.
SMP_Connect	Connects to a system that already resides in memory.
SMP_Expl	Gets the ASCII description of a specified identifier.
SMP_ExplAbs	Gets the absolute address of an ASCII description of a specified identifier.
SMP_ForceState	Forces the internal state configuration into a different state.
SMP_Free	Frees the memory allocated by a previous call to SMP_Connect.
SMP_GetInput	Finds events that can trigger transitions from the current state.
SMP_GetInputAll	Finds all events that can trigger transitions from the current state.
SMP_GetOutput	For internal API use only.
SMP_Init	Initializes the system.
*System*SMP_InitAll	Wraps one connecting function and all initialization functions except for the function that initializes global external variables.
SMP_InitGuardCallBack	Initializes the guard expression call-back function.

Table 30: Summary of the Uniform API functions

Uniform API function	Description
SMP_InitInstances	Initializes a number of instances of a system.
SMP_InitSignalQueue	Initializes the signal queue in a system.
SMP_Inquiry	Prepares for finding events that can trigger changes in the current state.
SMP_Machine	Returns the state machine index of a specified state.
SMP_Name	Gets the ASCII name of a specified identifier.
SMP_NameAbs	Gets the pointer to the ASCII name of a specified identifier.
SMP_NextState	For internal API use only.
SMP_NextStateChg	For internal API use only.
SMP_SetInstance	Sets the currently active instance of the system.
SMP_State	Returns the current state of a specified state machine.
SMP_StateAll	Returns the active state of all state machines.
*System*VSDeduct*	Deduces all the relevant action expressions on the basis of the given event, the internal current state vector, and the transitions in the Visual State system.
*System*VSElementExpl*	Gets the pointer to the ASCII explanation of the specified identifier.
*System*VSElementName*	Gets the pointer to the ASCII name of the specified identifier.
*System*VSGetCurrentStateTree	Copies the strings representing the current state tree into the buffer.
*System*VSGetMaxCurrentStateTree	Returns the needed size for VSGetCurrentStateTree buffer.
VSGetSignature*	Returns the signature for the project.
*System*VSInitAll*	Wraps all initialization functions for the system.

Table 30: Summary of the Uniform API functions

* The function is only generated if the appropriate Coder option has been enabled. For more information, see the individual function.

Descriptions of the Uniform API functions

The following pages give detailed reference information about each Uniform API function.

*System*SEM_InitExternalVariables

Syntax

```
#include "sdata.h"

/* Optionally */
#include "cext.h"

void systemSEM_InitExternalVariables (void)
```

Defined in

*System*Data.c

Description

This function initializes the external variables in the system and must be called together with the SMP_Init function.

The function is auto-generated by the Coder during the code generation of a system if any external variables are present, and the -iev1 Coder option has been set.

The function will be placed in the system data source file and declared external in the system data header file. Optionally, the function will be placed in the system external variable source file and declared external in the system external variable header file.

The name of the function will be prefixed with the name of the system source file.

This function is normally called by SMP_InitAll and VSInitAll functions, so normally you do not need to call this function.

Argument

None.

Return value

None.

Example

None.

*System*SEM_InitInternalVariables

Syntax

```
#include "sdata.h"
void systemSEM_InitInternalVariables (void)
```

Defined in

*System*Data.c

(optionally in *cext*.c)

Description — This function initializes the internal variables in the system and must be called together with the `SMP_Init` function.

The function is auto-generated by the Coder during the code generation of a system if any internal variables are present, and the `-iev1` Coder option has been set.

The function will be placed in the system data source file and declared external in the system data header file.

The name of the function will be prefixed with the name of system source file.

This function is automatically called by `SMP_InitAll` and `VSInit` functions, so normally you do not need to call this function.

Argument — None.

Return value — None.

Example — None.

*Project*SEM_InitPrjExternalVariables

Syntax

```
#include "gext.h"
void projectSEM_InitPrjExternalVariables (void)
```

Defined in — *gext*.c

Description — This function initializes the external variables in the project and must be called together with the `SMP_Init` function.

The function is auto-generated by the Coder during the code generation of a system if any external project variables are present, and the `-iev1` Coder option has been set.

The function will be placed in the project external variable file and declared external in the project external variable header file.

The name of the function will be prefixed with the name of the *gext*.c file.

Argument — None.

Return value — None.

Example — See *SMP_Connect*, page 640.

SMP_Action

Syntax

```
#include "project.h"
#define SMP_Action(Context, ActionNo)
                  (*VSAction[ActionNo])(Context)
```

Defined in

project.h

Description

This deprecated macro is provided for backward compatibility and should not be used. Instead use the `-vsdeduct1` Coder option, see *-vsdeduct*, page 752.

This is not a function, but a macro that is used in the same way as a function. The macro uses the `VSAction` function pointer table to call an action expression function.

Argument

`ActionNo`	The action expression index number.
`Context`	A pointer to a system context.

Return value

None.

Example

See *SMP_InitInstances*, page 652.

SMP_Connect

Syntax

```
#include "project.h"
unsigned char SMP_Connect(SEM_CONTEXT **Context, void *VSDdata)
```

Defined in

project.c

Description

This deprecated function is provided for backward compatibility and should not be used. Instead use the `-vsinitall1` Coder option, see *-vsinitall*, page 754.

This function connects to a binary system that already resides in memory.

Argument

`Context`	A pointer to a system context.
`VSDdata`	A pointer to the memory area where the system resides. The area must be an image of the binary system file including texts.

Return value

See:

SES_MEM_ERR, page 670

SES_NULL_PTR, page 671

SES_OKAY, page 671

SES_TYPE_ERR, page 672

Example

```
/* Header file for the generated VS System My_System. */
#include "My_System.h"

void Task (void)
{
  SEM_ACTION_EXPRESSION_TYPE ActionExpress;
  SEM_CONTEXT *Context;
  unsigned char cc;

  /*
    * Declare and initialize. The
    * reset event is SE_RESET.
    */
  SEM_EVENT_TYPE EventNo = SE_RESET;

  /* Initialize Visual State system My_System */
  if ((cc = My_SystemSMP_InitAll(&Context)) != SES_OKAY)
    ErrorHandling(cc);

  /*
   * If your project has external variables,
   * and you have chosen to initialize by function:
   * My_ProjectSEM_InitPrjExternalVariables();, where My_Project
   * is the name of the project.
   */
  While (1)
  {
    /* Start event deduction. */
    cc = My_SystemVSDeduct(Context, EventNo);
    If ((cc != SES_OKAY) && (cc != SES_FOUND))
      ErrorHandling(cc);
```

```
    /*
     * If you enabled SEM_NextStateChg:
     * if (cc==SES_FOUND)
     * {
     * use the information, that a state has
     * changed, for something...
     * }
     */
    /* Get new event and map it to VS System event. */
     MapEvent(&EventNo);
  }
}
```

SMP_Expl

Syntax

```
#include "project.h"
unsigned char SMP_Expl (SEM_CONTEXT *Context,
  unsigned char IdentType, SEM_EXPLANATION_TYPE IdentNo,
  char *Text, unsigned short MaxSize)
```

Defined in

project.c

Description

This deprecated function is provided for backward compatibility and should not be used. Instead use the function `VSElementExpl`, see *VSElementExpl*, page 616.

This function gets the UTF–8 description of the specified identifier.

Argument

`Context`	A pointer to a system context.
`IdentNo`	The index number of an identifier.
`IdentType`	The type of the identifier number, `EVENT_TYPE` or `STATE_TYPE`. Note that `ACTION_TYPE` is deprecated because there is no advantage to using it here.
`MaxSize`	The maximum length of the text including the `NULL` termination character.
`Text`	A pointer to a `char` pointer. If the function terminates successfully, the pointer points to the text that contains the name of the specified identifier.

Return value — See:

SES_FORMAT_ERR, page 670

SES_NULL_PTR, page 671

SES_OKAY, page 671

SES_RANGE_ERR, page 671

SES_TEXT_TOO_LONG, page 672

SES_TYPE_ERR, page 672

Example — See *SMP_GetInput*, page 646.

SMP_ExplAbs

Syntax

```
#include "project.h"
unsigned char SMP_ExplAbs (SEM_CONTEXT *Context,
                           unsigned char IdentType,
                           SEM_EXPLANATION_TYPE IdentNo,
                           char **Text)
```

Defined in — *project.c*

Description — This deprecated function is provided for backward compatibility and should not be used. Instead use the function `VSElementExpl`, see *VSElementExpl*, page 616.

This function gets the absolute address of an ASCII description of the specified identifier.

Argument

`Context`	A pointer to a system context.
`IdentNo`	The index number of an identifier.
`IdentType`	The type of the identifier number, `EVENT_TYPE` or `STATE_TYPE`. Note that `ACTION_TYPE` is deprecated because there is no advantage to using it here.
`Text`	A pointer to a `char` pointer. If the function terminates successfully, the pointer contains the absolute address of the name of the specified identifier.

Return value

See:

SES_NULL_PTR, page 671

SES_OKAY, page 671

SES_RANGE_ERR, page 671

SES_TYPE_ERR, page 672

Example

See *SMP_GetInputAll*, page 648.

SMP_ForceState

Syntax

```
#include "project.h"
unsigned char SMP_ForceState (SEM_Context *Context,
                              SEM_STATE_TYPE StateNo)
```

Defined in

project.c

Description

This function forces the internal state configuration into the specified state. This is useful if you want to reestablish the internal state configuration after a power failure of the target system. Before calling this function the first time after a power failure, the `SMP_InitAll` function should be called to initialize.

The state configuration established by calling `SMP_ForceState` must have been stored in EEPROM before the power failure.

Note: This function should be used with caution. The internal state configuration could be forced to a configuration that has not been verified.

Argument

`Context`	A pointer to a system context.
`StateNo`	Contains the state index number.

Return value

See:

SES_NULL_PTR, page 671

SES_OKAY, page 671

SES_RANGE_ERR, page 671

Example

```
/*
 * This function should only be called after a power failure
 * to reestablish the internal state variables.
 */
void PowerUp (void)
{
  SEM_STATE_TYPE StateNo;
  SEM_STATE_MACHINE_TYPE i;

  SEM_CONTEXT *Context;

  if (SystemSMPInitAll(Context) != SES_OKAY)
    ErrorHandling ();

  for (i = 0; i < VS_NOF_STATE_MACHINES; i++)
  {
    /* Get state configuration from EEPROM. */
    EEPROMState (i, &StateNo);
    if (SMP_ForceState (Context, StateNo) != SES_OKAY)
      ErrorHandling ();
  }
}
```

SMP_Free

Syntax

```
#include "project.h"
void SMP_Free (SEM_CONTEXT *Context)
```

Defined in

project.c

Description

This function frees the memory allocated by a previous call to `SMP_Connect`.

If `Context` is not equal to `NULL`, the memory allocated by the system and the context will be freed. If `Context` is `NULL`, nothing happens.

Argument

`Context`	A pointer to a system context.

Return value

None.

Example

```
void Change (void)
{
  SEM_CONTEXT *Context;

  /* Initialize System 1 */
  if (System1SMP_InitAll(&Context) != SES_OKAY)
    ErrorHandling();

  /* Use System 1 */
  PerformSystem1Deduction (Context);

  /* Free System 1 */
  SMP_Free (Context);

  /* Initialize System 2 */
  if (System2SMP_InitAll(&Context) != SES_OKAY)
    ErrorHandling();

  /* Use System 2 */
  PerformSystem2Deduction (Context);

  /* Free System 2 */
  SMP_Free (Context);
}
```

SMP_GetInput

Syntax

```
#include "project.h"
unsigned char SMP_GetInput (SEM_CONTEXT *Context,
  SEM_EVENT_TYPE *EventNo, SEM_EVENT_TYPE *EventList)
```

Defined in

project.c

Description

The function finds events that can trigger transitions or derive action expressions from the current state. All events are found by continuous calls to this function. Because the function will inquire events on the basis of the internal current state configuration, an event deduction should not be running.

Argument

Context	A pointer to a system context.

`EventList`	A pointer to an array that holds the event numbers to be inquired. `EventList` must be terminated with the symbol `EVENT_TERMINATION_ID`.
	If the pointer is `NULL`, all events are inquired.
`EventNo`	A pointer to the array in which to store the inquired event number.

Return value

See:

SES_FOUND, page 670

SES_NULL_PTR, page 671

SES_OKAY, page 671

SES_RANGE_ERR, page 671

Example

```
#define STRLEN 80

/* Used event definitions are found in System.h. */
const SEM_EVENT_TYPE KeyTable[] =
{
  E_KEY_F1,
  E_KEY_F2,
  E_KEY_F3,
  E_KEY_F4,
  E_KEY_F5,
  E_KEY_F6,
  E_KEY_F7,
  E_KEY_F8,
  E_KEY_F9,
  E_KEY_F10,
  E_KEY_F11,
  E_KEY_F12,
  EVENT_TERMINATION_ID
};

/* Print active keys. */
unsigned char PrintActiveKeys (SEM_CONTEXT *Context)
{
  char Str[STRLEN];
  unsigned char CC = SES_OKAY;
  SEM_EVENT_TYPE EventNo = EVENT_UNDEFINED;
```

```
  if ((CC = SMP_Inquiry (Context)) == SES_OKAY)
  {
    printf ("\nActive event number:");
    while ((CC = SMP_GetInput (Context, &EventNo, KeyTable))
      == SES_FOUND)
    {
      if (SMP_Expl (Context, EVENT_TYPE, EventNo, Str, STRLEN)
        == SES_OKAY)
        printf ("\n%s = %d", Str, EventNo);
        /*
         * Alternatively, call SMP_Name to get the name
         */
    }
  }
}
```

SMP_GetInputAll

Syntax

```
#include "project.h"
unsigned char SMP_GetInputAll (SEM_CONTEXT *Context,
  SEM_EVENT_TYPE *EventVector, SEM_EVENT_TYPE *EventList,
  SEM_EVENT_TYPE MaxSize)
```

Defined in

`project.c`

Description

This function finds all events that can trigger transitions or derive action expressions from the current state. All events are found by one call to this function. Because the function will inquire events on the basis of the internal current state configuration, an event deduction should not be running.

Argument

`Context`	A pointer to a system context.
`EventList`	A pointer to an array that holds the event numbers to be inquired. `EventList` must be terminated with the symbol `EVENT_TERMINATION_ID`. If the pointer is `NULL`, all events are inquired.
`EventVector`	A pointer to an array in which to store the inquired events.
`MaxSize`	The maximum length of the event vector including the symbol `EVENT_TERMINATION_ID`

Return value

See:

SES_BUFFER_OVERFLOW, page 669

SES_NULL_PTR, page 671

SES_OKAY, page 671

SES_RANGE_ERR, page 671

Example

```
#define STRLEN 80

/* Used event definitions are found in System.h. */
const SEM_EVENT_TYPE KeyTable[] =
{
  E_KEY_F1,
  E_KEY_F2,
  E_KEY_F3,
  E_KEY_F4,
  E_KEY_F5,
  E_KEY_F6,
  E_KEY_F7,
  E_KEY_F8,
  E_KEY_F9,
  E_KEY_F10,
  E_KEY_F11,
  E_KEY_F12,
  EVENT_TERMINATION_ID
};

/* Print active keys. */
unsigned char PrintActiveKeys (SEM_CONTEXT *Context)
{
  char Str[STRLEN];
  unsigned char CC = SES_OKAY;
  SEM_EVENT_TYPE EventList[13];
  int i;
```

```
  if ((CC = SMP_Inquiry (Context)) == SES_OKAY)
  {
    printf ("\nActive event number:");
    if ((CC = SMP_GetInputAll (Context, &EventNo, KeyTable, 13))
      == SES_FOUND)
    {
      i = 0;
      while (EventList[i] != EVENT_TERMINATION_ID)
      {
        /* Alternatively, call SMP_NameAbs to get the name. */
        if (SMP_ExplAbs (Context, EVENT_TYPE, EventList[i], &Str)
          == SES_OKAY)
          printf ("\n%s = %d", Str, EventList[i++]);
      }
    }
  }
}
```

See also *SMP_Inquiry*, page 655.

SMP_GetOutput

Description

This function is for internal use only.

SMP_Init

Syntax

```
#include "project.h"
unsigned char SMP_Init (SEM_CONTEXT *Context)
```

Defined in

`project.c`

Description

This deprecated function is provided for backward compatibility and should not be used. Instead use the `-vsinitall1` Coder option, see *-vsinitall*, page 754.

This function initializes the system.

`SMP_Init` is called automatically by `SEM_InitAll`, which means that you should normally not need to call `SMP_Init`.

Note: The recommended way to connect and initialize is to call the *`System`*`VSInitAll` function.

Argument

`Context`	A pointer to a system context.

Return value

See:

SES_NULL_PTR, page 671

SES_OKAY, page 671

Example

None.

*System*SMP_InitAll

Syntax

```
#include "sdata.h"
void systemSMP_InitAll (SEM_CONTEXT** Context)
```

Defined in

`SystemData.c`

Description

This function wraps one connecting function and all initialization functions except for the function that initializes global external variables (see *ProjectSEM_InitPrjExternalVariables*, page 639). The function calls the following functions in the listed order, provided that they exist:

```
SMP_Connect
SMP_Init
SystemSMP_InitExternalVariables
SystemSMP_InitInternalVariables
SMP_InitSignalQueue
SMP_InitInstances
SMP_InitGuardCallBack
SMP_InitSignalDBCallBack
```

The function must be enabled by the Coder command line option `-seminitall1` or the corresponding GUI option.

Argument

`Context`	A pointer to a pointer to a system context. On returning from the function, the pointer will point to a system context.

Return value

All possible return values returned by the wrapped functions.

Example

See *SMP_GetOutput*, page 650.

SMP_InitGuardCallBack

Syntax

```
#include "project.h"
void SMP_InitGuardCallBack (SEM_CONTEXT *Context,
  unsigned char (*Guard[])(SEM_CONTEXT *Context))
```

Defined in

project.c

Description

This deprecated function is provided for backward compatibility and should not be used. Instead use the -vsinitall Coder option, see *-vsinitall*, page 754.

This function initializes the guard expression call-back function. Call this function after the SMP_Connect function if the system contains guard expressions.

SMP_InitGuardCallBack is called automatically by *System*SEM_InitAll, which means that you should normally not need to call SMP_InitGuardCallBack.

Argument

Context	A pointer to a system context.
Guard	A pointer to the system guard expression function pointer table.

Return value

None.

Example

None.

SMP_InitInstances

Syntax

```
#include "project.h"
unsigned char SMP_InitInstances (SEM_CONTEXT *Context)
```

Defined in

project.c

Description

This function initializes a number of instances of a system. The instance is handled in pseudo-parallel using the SMP_SetInstance function. The actual number of instances is determined by the information in the system.

If the function has already been called, any previous instances are deallocated and a new set is allocated.

SMP_InitInstances is called automatically by *System*SEM_InitAll, which means that you should normally not need to call SMP_InitInstances.

Argument

`Context`	A pointer to a system context.

Return value

See:

SES_MEM_ERR, page 670

SES_NULL_PTR, page 671

SES_OKAY, page 671

Example

```
unsigned char Instance (SEM_CONTEXT *Context,
  SEM_EVENT_TYPE EventNo, SEM_INSTANCE_TYPE InstanceNo)
{
  /* Declare action expression variable. */
  SEM_ACTION_EXPRESSION_TYPE ActionExpress;

  /* Declare completion code */
  unsigned char cc;

  /* Set active instance. */
  if (SMP_SetInstance (Context, InstanceNo) != SES_OKAY)
    return (FALSE);
  cc = SystemVSDeduct (Context, EventNo);
  if ((cc != SES_OKAY) && cc != SES_FOUND)
    return (FALSE);
  return (TRUE)
}


void Task (void)
{
  SEM_CONTEXT Context;
  SEM_INSTANCE_TYPE InstanceNo = 0;

  /*
   * Declare and initialize. In this case the
   * reset event is SE_RESET.
   */
  SEM_EVENT_TYPE EventNo = SE_RESET;
```

```
  if (SystemSMPInitAll(&Context) != SES_OKAY)
    ErrorHandling ();

  for (InstanceNo = 0; InstanceNo < VS_NOF_INSTANCES;
    InstanceNo++)
  {
    Instance (Context, EventNo, InstanceNo);
  }

  /* Do forever. */
  while (1)
  {
    /*
     * Get new event and map it to VS System events and
     * instance.
     */
    MapEvent (&EventNo, &InstanceNo);
    /* Process the event. */
    if (Instance (Context, EventNo, InstanceNo) != TRUE)
      ErrorHandling ();
  }
}
```

See also

The Visual State system, page 123.

SMP_InitSignalQueue

Syntax

```
#include "project.h"
void SMP_InitSignalQueue (SEM_CONTEXT *Context)
```

Defined in

project.c

Description

This function initializes the signal queue in a system and must be called together with the `SMP_Init` function. The function will only be available if the signal queue is enabled and the system contains signals.

`SMP_InitSignalQueue` is called automatically by `SMP_Connect`, so unless the system needs to be reinitialized, this function does not need to be called.

`SMP_InitSignalQueue` is called automatically by *System*`SEM_InitAll`, which means that you should normally not need to call `SMP_InitSignalQueue`.

Argument

`Context`	A pointer to a system context.

Return value

None.

Example

None.

SMP_Inquiry

Syntax

```
#include "project.h"
unsigned char SMP_Inquiry (SEM_CONTEXT *Context)
```

Defined in

project.c

Description

This function prepares for finding events that can trigger changes in the current state. All events are found by continuous calls to the function `SMP_GetInput` or one call to `SMP_GetInputAll`.

As the function will inquire events on the basis of the internal current state configuration, `SMP_Inquiry` can only be used if the previously called function is one of these:

- `SMP_Connect`
- `SMP_Init`
- `SMP_NextState`
- `SMP_NextStateChg`

Argument

`Context`	A pointer to a system context.

Return value

See:

SES_ACTIVE, page 669

SES_NULL_PTR, page 671

SES_OKAY, page 671

Example

```
#define STRLEN 80

/* Print active events */
unsigned char PrintActiveEvents(SEM_Context *Context)
{
  char Str[STRLEN];
  unsigned char CC = SES_OKAY;
  SEM_EVENT_TYPE EventNo = EVENT_UNDEFINED;

  if ((CC = SMP_Inquiry(Context)) == SES_OKAY)
  {
    printf("\nActive event numbers:");
    while ((CC = SMP_GetInput(Context, &EventNo, NULL))
      == SES_FOUND)
    {
      if (SMP_Name(Context, EVENT_TYPE, EventNo, Str, STRLEN)
          == SES_OKAY)
        printf("\n%s = %d", Str, EventNo);
        /*
         * Alternatively to using SMP_Name:
         * {
         *   const char *pName;
         * if((cc = SMP_NameAbs(Context, EVENT_TYPE, EventNo,
               &pName)) == SES_OKAY)
         *   printf("\n%s = %d", pName, EventNo);
         * }
         */
    }
  }
  return (CC);
}
```

See also *SMP_GetInput*, page 646.

SMP_Machine

Syntax

```
#include "project.h"
unsigned char SMP_Machine (SEM_Context *Context,
                          SEM_STATE_TYPE StateNo,
                          SEM_STATE_MACHINE_TYPE *StateMachineNo)
```

Defined in

project.c

Description

This function returns the state machine index of the specified state.

Argument

`Context`	A pointer to a system context.
`StateMachineNo`	A pointer for storing the state machine index number found for the specified state.
`StateNo`	The state machine index number.

Return value

See:

SES_FOUND, page 670

SES_NULL_PTR, page 671

SES_RANGE_ERR, page 671

Example

```
/*
  * The function is used for turning on/off a standby LED
  * The function must be called, just after SMP_NextState is
  * called.
  */
unsigned char CheckStandby (SEM_Context *Context)
{
  unsigned char CC;
  SEM_STATE_TYPE StateNo;
  SEM_STATE_MACHINE_TYPE StateMachine;

  /* State STATE_STANDBY defined in SystemData.h file. */
  if ((CC = SMP_Machine (Context, STATE_STANDBY, &StateMachine))
    == SES_FOUND)
  {
    if (StateNo == STATE_STANDBY)
      StandbyLED = TRUE;
    else
      StandbyLED = FALSE;
  }
  return (CC);
}
```

SMP_Name

Syntax

```
#include "project.h"
unsigned char SMP_Name (SEM_CONTEXT *Context,
                        unsigned char IdentType,
                        SEM_EXPLANATION_TYPE IdentNo,
```

```
char *Text, unsigned short MaxSize)
```

Defined in *project*.c

Description

This deprecated function is provided for backward compatibility and should not be used. Instead use the function *System*VSElementName, see *SystemVSElementName*, page 666.

This function gets the ASCII name of the specified identifier and can only be used (and compiled) when at least one type of name is included in the system.

Argument

Context	A pointer to a system context.
IdentNo	The index number of an identifier.
IdentType	The type of the identifier, which can be one of: EVENT_TYPE, STATE_TYPE, or ACTION_TYPE.
MaxSize	The maximum length of the text, including the NULL termination character.
Text	A pointer to a text string. If the function terminates successfully, the text string contains the name of the specified identifier.

Return value

See:

SES_FORMAT_ERR, page 670

SES_NULL_PTR, page 671

SES_OKAY, page 671

SES_RANGE_ERR, page 671

SES_TEXT_TOO_LONG, page 672

SES_TYPE_ERR, page 672

Example

See *SMP_Inquiry*, page 655.

SMP_NameAbs

Syntax

```
#include "project.h"
unsigned char SMP_NameAbs (SEM_CONTEXT *Context,
```

```
unsigned char IdentType,
SEM_EXPLANATION_TYPE IdentNo,
char **Text)
```

Defined in

project.c

Description

This deprecated function is provided for backward compatibility and should not be used. Instead use the function `SystemVSElementName`, see *SystemVSElementName*, page 666.

This function gets the pointer to the ASCII name of the specified identifier.

Argument

`Context`	A pointer to a system context.
`IdentNo`	Must contain the index number of an identifier.
`IdentType`	Must contain the type of the identifier which can be `EVENT_TYPE`, `STATE_TYPE`, or `ACTION_TYPE`.
`Text`	Must be a pointer to a `char` pointer. If the function terminates successfully, the pointer points to the text that contains the name of the specified identifier.

Return value

See:

SES_NULL_PTR, page 671

SES_OKAY, page 671

SES_RANGE_ERR, page 671

SES_TYPE_ERR, page 672

Example

```
SEM_EVENT_TYPE eventNo;
/* Get next event from the queue */
if (DEQ_RetrieveEvent(&eventNo) != UCC_QUEUE_EMPTY)
{
  const char *pName;
  unsigned char cc;
  if ((cc = SMP_NameAbs(pSEMContext, EVENT_TYPE, eventNo,
            &pName)) != SES_OKAY)
    handleError(cc);
  printf("Event '%s' sent to the system.", pName);
}
```

SMP_NextState

Description

This function is for internal use only.

SMP_NextStateChg

Description

This function is for internal use only.

SMP_SetInstance

Syntax

```
#include "project.h"
unsigned char SMP_SetInstance (SEM_CONTEXT *Context,
                               SEM_INSTANCE_TYPE Instance)
```

Defined in

project.c

Description

This function sets the currently active instance of the system. The instance remains active until the next call to this function. The function may only be called between completed macrosteps, not in the middle of a macrostep.

Argument

`Context`	A pointer to a system context.
`Instance`	The instance to be handled.

Return value

See:

SES_ACTIVE, page 669

SES_NULL_PTR, page 671

SES_OKAY, page 671

SES_RANGE_ERR, page 671

Example

See *SMP_InitInstances*, page 652.

SMP_State

Syntax

```
#include "project.h"
unsigned char SMP_State (SEM_CONTEXT *Context,
                         SEM_STATE_MACHINE_TYPE StateMachineNo,
```

```
                              SEM_STATE_TYPE *StateNo)
```

Defined in *project.c*

Description This function returns the current state of the specified state machine.

Argument

Context	A pointer to a system context.
StateMachineNo	The state machine number.
StateNo	A pointer to the location in which to store the current state of the specified state machine.

Return value See:

SES_FOUND, page 670

SES_NULL_PTR, page 671

SES_RANGE_ERR, page 671

Example

```
void Task (void)
{
  SEM_STATE_TYPE StateNo = STATE_UNDEFINED;
  SEM_STATE_MACHINE_TYPE i;

  SEM_CONTEXT *Context;

  /*
   * Declare and initialize event variable.
   * In this case the reset event is SE_RESET.
   */
  SEM_EVENT_TYPE EventNo = SE_RESET;

  if (SystemSMP_InitAll(&Context) != SES_OKAY)
    ErrorHandling();

  /* Do forever. */
  while (1)
  {
    unsigned char cc = SystemVSDeduct(Context, EventNo);
    if (cc != SES_OKAY && cc != SES_FOUND)
      ErrorHandling ();
```

```
    for (i = 0; i < VS_NOF_STATE_MACHINES; i++)
    {
      if (SMP_State (Context, i, &StateNo) != SES_FOUND)
        printf ("\nState machine %d is in undefined state", i);
      else
        /* Print state machine number and state number. */
        printf ("\nState machine %d: state %d", i, StateNo);
    }

    /* Get new event and map it to System events. */
    MapEvent (&EventNo);
  }
}
```

SMP_StateAll

Syntax

```
#include "project.h"
unsigned char SMP_StateAll (SEM_Context *Context,
                            SEM_STATE_TYPE *StateVector,
                            SEM_STATE_MACHINE_TYPE MaxSize)
```

Defined in

project.c

Description

This function returns the active state of all state machines.

Argument

Context	A pointer to a system context.
MaxSize	The maximum size of the destination array. Normally the size will be equal to the number of state machines. The array is not terminated.
StateVector	A pointer to an array in which to store the current state configuration.

Return value

See:

SES_BUFFER_OVERFLOW, page 669

SES_FOUND, page 670

SES_NULL_PTR, page 671

Example

```
void Task (void)
{
  SEM_STATE_TYPE StateList[VS_NOF_STATE_MACHINES];
  SEM_STATE_MACHINE_TYPE i;

  SEM_Context *Context;

  /*
   * Declare and initialize. In this case the
   * reset event is SE_RESET.
   */
  SEM_EVENT_TYPE EventNo = SE_RESET;

  if (SMP_Connect(&Context, &System) != SES_OKAY)
    ErrorHandling ();

  /* Do forever. */
  while (1)
  {
    unsigned char cc = SystemVSDeduct(Context, EventNo);
    if (cc != SES_OKAY && cc != SES_FOUND)
      ErrorHandling ();

    if (SMP_StateAll (Context, StateList, VS_NOF_STATE_MACHINES)
      != SES_FOUND)
      printf ("\nCannot access states.");
    else
    {
      /* Print state machine number and state number. */
      for (i = 0; i < VS_NOF_STATE_MACHINES; i++)
        printf ("\nState machine %d: state %d", i,
          StateList[i]);
    }

    /* Get new event and map it to System events. */
    MapEvent (&EventNo);
  }
}
```

*System*VSDeduct

Syntax

```
VS_UINT8 SystemVSDeduct(SEM_CONTEXT  * Context, SEM_EVENT_TYPE
EventNo, ...);
```

Defined in

*System*Data.c

Description

This function deduces all the relevant action expressions on the basis of the given event, the internal current state configuration, and the transitions in the Visual State system. All the relevant action expressions are then called and all the next states are changed.

Note: This function is only available if you have enabled the `-vsdeduct1` Coder option.

Argument

`Context`	A pointer to a system context.
`EventNo`	The event number to be processed. If at least one event has parameters, the function call must include one argument for each parameter declared in the parameter list for each event.

Return value

See:

SES_CONTRADICTION, page 669

SES_FOUND, page 670

SES_OKAY, page 671

SES_RANGE_ERR, page 671

SES_SIGNAL_QUEUE_FULL, page 671

Example

```
/* do forever */
while (1)
{
  cc = systemnameVSDeduct(Context, eventNo);
  /*
   * If you enabled the semnextstatechg Coder option
   * if (cc == SES_FOUND)
   * {
   * react to a change in some state
   * }
  */
  if (cc != SES_OKAY && cc != SES_FOUND)
    handleError(cc);
  /* Get new event and map it to VS system events */
  MapEvent (&eventNo);
}
```

*System*VSElementExpl

Syntax

```
unsigned char SystemVSElementExpl (SEM_CONTEXT * Context,
unsigned char IdentType, SEM_EXPLANATION_TYPE IdentNo, char const
** Text)
```

Defined in

*System*Data.c

Description

This function gets a pointer to the explanation of the specified identifier. You must also enable generation of the explanations you want to get. For example, set the Coder options `-vselementexpl1` and `-txte2`. This will enable the function, and enable generating explanations for events.

Argument

Context	A pointer to a system context.
IdentType	Contains the type of the identifier, EVENT_TYPE or STATE_TYPE.
IdentNo	Contains the index number of an identifier.
Text	Contains the address of a pointer to a text string. If the function terminates successfully, the text pointer contains the address of the explanation of the specified identifier.

Return value

See:

SES_OKAY, page 671

SES_RANGE_ERR, page 671

SES_TYPE_ERR, page 672

Example

```
SEM_EVENT_TYPE eventNo;
/* Get next event from the queue */
if (DEQ_RetrieveEvent(&eventNo) != UCC_QUEUE_EMPTY)
{
  char const *pExpl;
  unsigned char cc;
  if ((cc = SystemVSElementExpl(pSEMContext,
                                EVENT_TYPE,
                                eventNo, &pExpl)) != SES_OKAY)
    handleError(cc);
  printf("Event with the explanation '%s' sent to the system.",
pExpl);
}
```

*System*VSElementName

Syntax

```
unsigned char systemnameVSElementName (SEM_CONTEXT * Context,
unsigned char IdentType, SEM_EXPLANATION_TYPE IdentNo, char const
** Text)
```

Defined in

`System`*Data.c*

Description

This function gets a pointer to the name of the specified identifier. You must also enable generation of the names you want to get. For example, set the Coder options `-vselementname1` and `-txte1`. This will enable the function, and enable generating names for events.

Argument

`Context`	A pointer to a system context.
`IdentType`	Contains the type of the identifier, `EVENT_TYPE` or `STATE_TYPE`.
`IdentNo`	Contains the index number of an identifier.
`Text`	Contains the address of a pointer to a text string. If the function terminates successfully, the text pointer contains the address of the name of the specified identifier.

Return value

See:

SES_OKAY, page 671

SES_RANGE_ERR, page 671

SES_TYPE_ERR, page 672

Example

```
SEM_EVENT_TYPE eventNo;
/* Get next event from the queue */
if (DEQ_RetrieveEvent(&eventNo) != UCC_QUEUE_EMPTY)
{
  char const *pName;
  unsigned char cc;
  if ((cc = SystemVSElementName(pSEMContext,
                                EVENT_TYPE,
                                eventNo,
                                &pName)) != SES_OKAY)
    handleError(cc);
  printf("Event '%s' sent to the system.", pName);
}
```

*System*VSGetCurrentStateTree

Syntax

```
VSResult SystemVSGetCurrentStateTree (SEM_CONTEXT * Context, char
                                      * buf, size_t const bufSize)
```

Defined in

System.c

Description

This function copies the strings that represent the current state tree into the buffer. Each entry ends with a semicolon.

Argument

`Context`	A pointer to a system context.
`buf`	A pointer to a buffer.
`bufSize`	The size of the buffer `buf`.

Return value

See:

SES_OKAY, page 671

SES_TEXT_TOO_LONG, page 672

Example

None.

*System*VSGetMaxCurrentStateTree

Syntax

```
size_t SystemVSGetMaxCurrentStateTree (SEM_CONTEXT * Context)
```

Defined in

System.c

Description

This function returns the required size of the `VSGetCurrentStateTree` buffer.

Argument

`Context`	A pointer to a system context.

Return value

The required size of the `VSGetCurrentStateTree` buffer.

Example

None.

VSGetSignature

Syntax	`char const *VSGetSignature(void)`
Defined in	*`System`*`Data.c`
Description	This function returns the signature for the project.
Argument	None.
Return value	A pointer to the signature.
Example	None.

*System*VSInitAll

Syntax	`void `*`System`*`VSInitAll(SEM_CONTEXT * Context)`
Defined in	*`System`*`.c`
Description	This function wraps all initialization functions for the system and calls them. This function must be enabled by the Coder option `-vsinitall1`.
Argument	`Context` A pointer to a system context.
Return value	None.
Example	None.

Uniform API return codes

The following pages give detailed reference information about each UniformAPI return code.

SES_ACTIVE

Return code
`SES_ACTIVE`

Description
The completion code covers one of the following:

- An event deduction is started while an event inquiry is active. All inquired events have not been returned by the function `SMP_GetInput`.
- An event inquiry is started while an event deduction is active. All deduced action expressions have not been returned by the function `SMP_GetOutput` and the `SMP_NextState` has not been called to finish the event deduction.

Solution
The completion code is a warning and the application might have to be rewritten. An event inquiry and an event deduction should not be active at the same time.

SES_BUFFER_OVERFLOW

Return code
`SES_BUFFER_OVERFLOW`

Description
A destination buffer cannot hold the number of items found.

Solution
Call the function with an extended buffer as destination.

SES_CONTRADICTION

Return code
`SES_CONTRADICTION`

Description
A contradiction has been detected between two states in a state machine.

Solution
Check the system with the Validator or the Verificator.

SES_EMPTY

Return code SES_EMPTY

Description No events have been given to the *System*VSDeduct function before calling this function.

Solution Call the *System*VSDeduct function with an event number.

SES_FORMAT_ERR

Return code SES_FORMAT_ERR

Description The data in the Visual State system has an incorrect format.

Solution Check that the correct version of the Coder has been used when generating the files.

SES_FOUND

Return code SES_FOUND

Description The called function has returned an identifier index number.

Solution Process the returned identifier index number. If the function SMP_GetInput or SMP_GetOutput was called, the function can be called again to find more events or action expressions.

SES_MEM_ERR

Return code SES_MEM_ERR

Description There has been an error while allocating memory for the system.

Solution

- Free some memory on the host computer
- Use a large data memory model.

SES_NULL_PTR

Return code `SES_NULL_PTR`

Description A null pointer has been given to the function instead of a valid `SEM_CONTEXT` pointer.

Solution Call the function with a valid `SEM_CONTEXT` pointer.

SES_OKAY

Return code `SES_OKAY`

Description Function performed successfully.

Solution Not applicable.

SES_RANGE_ERR

Return code `SES_RANGE_ERR`

Description A reference is being made to an identifier that does not exist. Note that the first index number is 0. If the Visual State system has 4 identifiers of the same type, and a function is called with a parameter value equal to 4, the function will return an `SES_RANGE_ERR` error. In this case the highest permitted variable value is 3.

Solution Your application can check the index parameters with one of the following variables in the `SEM_Context` structure (in the `SEMLibE.h` file):

```
nNofEvents
nNofStates
nNofActionFunctions
nNofStateMachines
```

SES_SIGNAL_QUEUE_FULL

Return code `SES_SIGNAL_QUEUE_FULL`

Description The signal queue is full.

Solution Increase the maximum signal queue size in your system.

SES_TEXT_TOO_LONG

Return code	`SES_TEXT_TOO_LONG`
Description	The requested text is longer than the specified maximum length.
Solution	Increase the maximum length.

SES_TYPE_ERR

Return code	`SES_TYPE_ERR`
Description	A text function has been called with the wrong identifier type, or the specified text is not included in the system.
Solution	Use the identifier type symbols (`EVENT_TYPE`, `STATE_TYPE` or `ACTION_TYPE`) defined in the `SEMLibE.h` file. Include wanted text in your system.

The Visual State Classic Coder

- Introduction to the Visual State Classic Coder
- Graphical environment for the Classic Coder
- SEM type identifiers
- Transition rule data format

Introduction to the Visual State Classic Coder

Learn more about:

- *Briefly about the Visual State Classic Coder*, page 673

BRIEFLY ABOUT THE VISUAL STATE CLASSIC CODER

There are two Visual State Coders to use for generating code from your state machine models for a specific API. For more information about code generation and the APIs, see *Code generation*, page 457.

Before you start the code generation, specify Coder options in the **Classic Coder Options** dialog box. Start the code generation by choosing **Project>Code generate** in the Navigator.

For a description of the Visual State Hierarchical Coder, see *The Visual State Hierarchical Coder*, page 493.

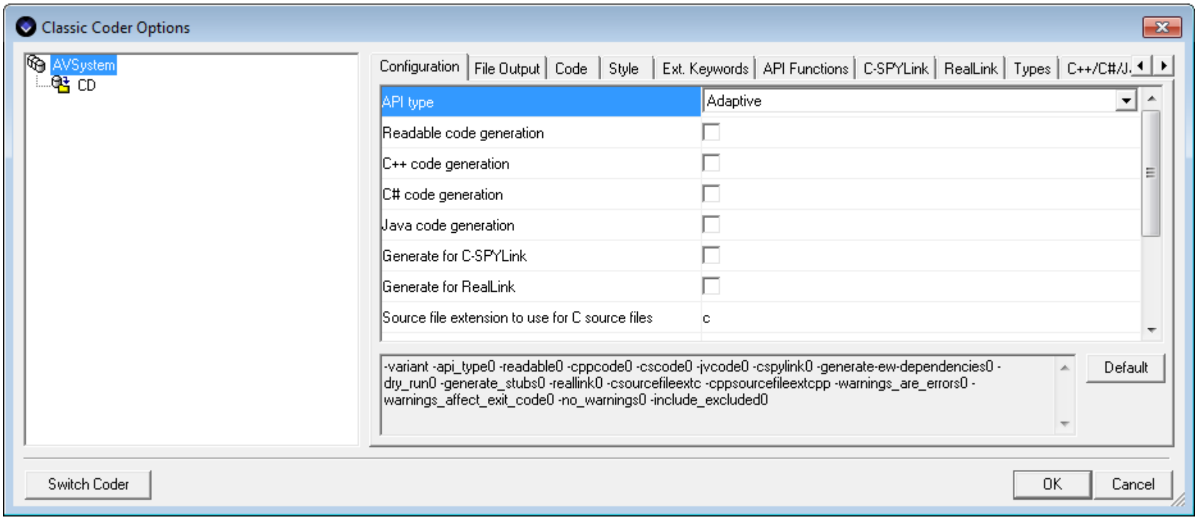
Graphical environment for the Classic Coder

Reference information about:

- *Classic Coder Options dialog box*, page 674

Classic Coder Options dialog box

The **Classic Coder Options** dialog box is available from the **Project** menu in the Navigator.



Use this dialog box to set options for code generation. Which options you can set depends on whether you are setting options on project level or on system level. Select either the project or a system in the pane to the left.

Use the **Switch Coder** button to switch from the Classic Coder to the Hierarchical Coder and back again.

For a description of an option, right-click it or select it and press Shift+F1.

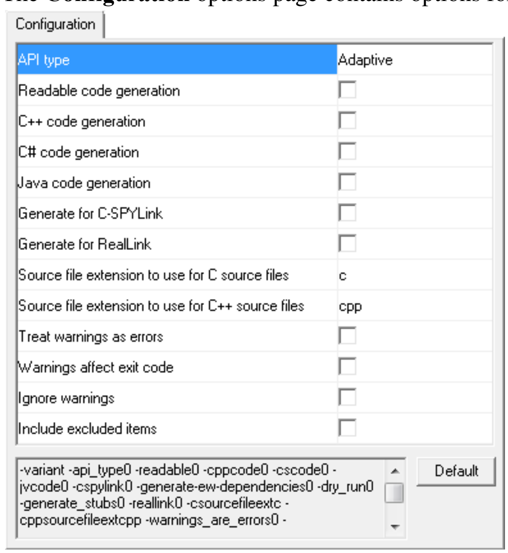
You can set options on these tabbed pages:

- *Classic Coder Options dialog box : Configuration*, page 675
- *Classic Coder Options dialog box : File Output*, page 677
- *Classic Coder Options dialog box : Code*, page 679
- *Classic Coder Options dialog box : Style*, page 682
- *Classic Coder Options dialog box : Extended Keywords*, page 684
- *Classic Coder Options dialog box : Names*, page 686
- *Classic Coder Options dialog box : API Functions*, page 689
- *Classic Coder Options dialog box : C++/C#/Java*, page 690
- *Classic Coder Options dialog box : Readable Code*, page 691

- *Classic Coder Options dialog box : C-SPYLink*, page 692
- *Classic Coder Options dialog box : RealLink*, page 694
- *Classic Coder Options dialog box : Types*, page 696
- *Classic Coder Options dialog box : MISRA*, page 697

Classic Coder Options dialog box : Configuration

The **Configuration** options page contains options for general configuration.



Use this page to make configuration settings for the Classic Coder. The display area under the options shows the resulting command line for the code generation.

This options page can only be accessed on project level.

API type

Specify the runtime API to use for code generation. For more information, see *The Visual State APIs*, page 459.

Choose between:

Adaptive

The Adaptive API is optimized for the data size of each system and has a copy of the API functions for each system.

Uniform

The Uniform API uses one shared API for all systems and uses the same data sizes for all systems.

Readable code generation

Generates readable code.

C++ code generation

Generates C++ code.

C# code generation

Generates C# code.

Java code generation

Generates Java code.

Generate for C-SPYLink

Generates code that can be debugged using C-SPYLink.

Generate for RealLink

Generates code that can be debugged using RealLink.

Source file extension to use for C source files

Type the filename extension that IAR Visual State shall use for generated C language source files.

Source file extension to use for C++ source files

Type the filename extension that IAR Visual State shall use for generated C++ language source files.

Treat warnings as error

Makes the Classic Coder treat all warnings as errors. If the Coder encounters an error, no code is generated.

Warnings affect exit code

By default, the exit code is not affected by warnings, because only errors produce a non-zero exit code. With this option, warnings will also generate a non-zero exit code.

Ignore warnings

By default, the Classic Coder issues warnings. Select this option to disable all warnings.

Include excluded items

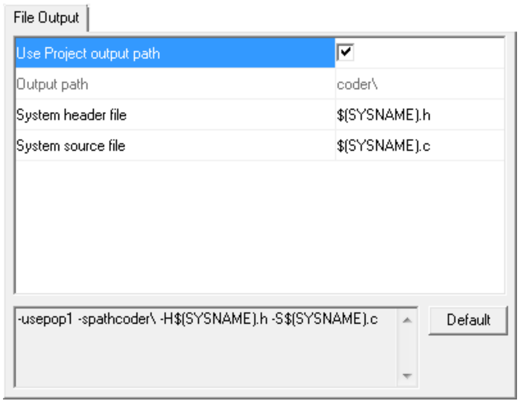
Makes the Classic Coder generate code also for graphical objects marked as excluded in the Designer.

Default

Restores the options to their default settings.

Classic Coder Options dialog box : File Output

The **File Output** options page contains options for file output from code generation.



Use this page to make file output settings for the Classic Coder. The display area under the options shows the resulting command line for the code generation. For C# and Java code generation, the output file names cannot be customized. They will be constructed from the name of the project or system, and have the filename extension `.cs` or `.java`.

This options page can be accessed on both project level and system level. Exactly which options you can set depends on the level you are setting options on.

Use project output path

Makes the Classic Coder use the same output path for system files as the path specified for all project files. This option can only be set on system level.

Output path

Specify the output path for all generated project or system files, respectively. If the path does not exist, it is created. The path can be relative. This option can be set on both project level and system level. For Java output, it might be a good idea to make sure that the output path is aligned with the package name.

System header file

Specify the name of the header file that contains system-level model declarations. The name used by default is `System.h`. This option can only be set on system level.

System source file

Specify the name of the source file that contains system-level model definitions. The name used by default is based on the name of the system, with a filename extension of either `c`, `cpp`, `cs`, or `java`. For C or C++ code, you might have to modify the filename extension manually. This option can only be set on system level.

Report file

Specify a name for a report file to contain information about the project, option settings, model characteristics, statistics, and a summary of the code generation. By default, no report file is generated. This option can only be set on project level.

Project source file

Specify the name of the source file that contains project-level model definitions. The name used by default is based on the name of the project, with a filename extension of either `c`, `cpp`, `cs`, or `java`. For C or C++ code, you might have to modify the filename extension manually. This option can only be set on project level.

Project header file

Specify the name of the header file that contains project-level model declarations. The name used by default is `Project.h`. This option can only be set on project level.

File that will be included verbatim in each generated source file

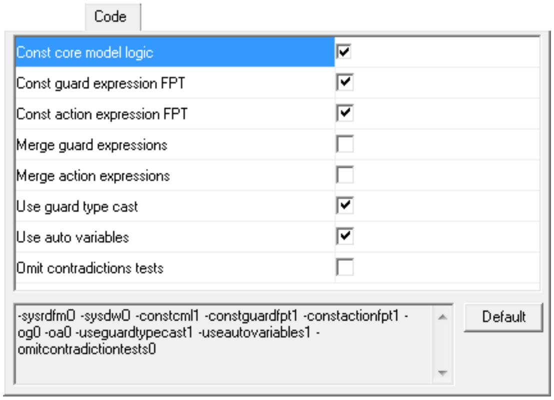
Specify the name for a file to include in every generated source file. This option can only be set on project level.

Default

Restores the options to their default settings.

Classic Coder Options dialog box : Code

The **Code** options page contains options for the actual code generation.



Use this page to make code settings for the Classic Coder. The display area under the options shows the resulting command line for the code generation.

This options page can be accessed on both project level and system level. Exactly which options you can set depends on the level you are setting options on.

External variable initialization

Specify how to initialize external variables. This option can only be set on project level.

Choose between:

By definition

Initializes variables along with their definition.

Both

Initializes variables in a function and by definition.

Internal variable initialization

Specify how to initialize internal variables. This option can only be set on project level.

Choose between:

By definition

Initializes variables along with their definition.

Both

Initializes variables in a function and by definition.

Functional expression handling

Specify how to handle functional expressions (guard expressions and action expressions). This option can only be set on project level.

Choose between:

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.

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 alternative settings efficiently.

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 can recognize the switch-case construct and convert it into a jump table, this might be the most efficient setting.

Optimize states and state machines

Optimizes states and state machines. Any state machine with a single state is optimized away to reduce the number of states, state machines, and the size of the core model logic. This option can only be set on project level.

Generate digital signature

Includes a digital signature in the generated code. This option can only be set on project level.

Generate time and version

Prevents accidentally mixing files generated by multiple code generations. This option can only be set on project level.

Use heap memory

Makes the Coder-generated code 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. This option can only be set on project level.

Automatic entry function call

Specify the name of a predefined function to call whenever a state is entered. This can help you debug the state machine. This option can only be set on project level.

Automatic exit function call

Specify the name of a predefined function to call whenever a state is exited. This can help you debug the state machine. This option can only be set on project level.

Const core model logic

Defines the core model logic as a `const` variable. This option should only be deselected in exceptional cases, for example, when the target controller has sufficient and fast RAM, and speed is of the highest importance. This option can only be set on system level.

Const guard expression FPT

Defines the guard expression function pointer table as a `const` variable. This option should only be deselected in exceptional cases, for example, when the target controller has sufficient and fast RAM, and speed is of the highest importance. This option can only be set on system level.

Const action expression FPT

Defines the action expression function pointer table as a `const` variable. This option should only be deselected in exceptional cases, for example, when the target controller has sufficient and fast RAM, and speed is of the highest importance. This option can only be set on system level.

Merge guard expressions

Merges guard expressions. This might increase execution speed because multiple guard expressions associated with a single transition are generated as a compound statement in the code. The drawback is that one and the same guard expression might be generated multiple times if constructs such as entry reactions, exit reactions, or history states are used. Deselecting this option might generate smaller code.

This option can only be set on system level.

Merge action expressions

Merges action expressions. This might increase execution speed because multiple action expressions associated with a single transition are generated as a compound statement in the code. The drawback is that one and the same action expression might be generated multiple times if constructs such as entry reactions, exit reactions, or history states are used. Deselecting this option might generate smaller code.

This option can only be set on system level.

Use guard type cast

Uses guard type casts. This option can only be set on system level.

Use auto variables

Allows auto variables in the generated API code. This might make the API code faster but it can also lead to increased stack usage. This option can only be set on system level.

Omit contradiction tests

Disables the generation of contradiction test code for each transition. This option should only be selected 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 readable code and table-based code. See also *Size of generated readable code*, page 462.

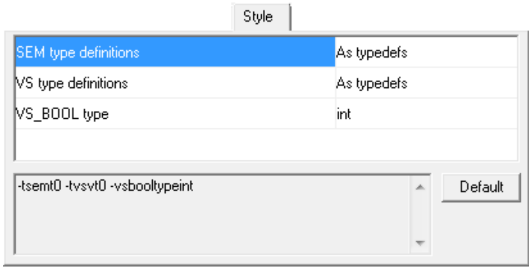
This option can only be set on system level.

Default

Restores the options to their default settings.

Classic Coder Options dialog box : Style

The **Style** options page contains options for style.



Use this page to make style settings for the Classic Coder. The display area under the options shows the resulting command line for the code generation.

This options page can only be accessed on project level.

SEM type definitions

Specify how to make SEM type definitions.

Choose between:

As typedefs

Uses the `typedef` keyword for type definitions. This is the preferred setting because it helps the compiler to do type checking.

As macros

Uses the `#define` keyword for type definitions. Use this setting if the compiler cannot determine that two type definitions actually are the same.

VS type definitions

Specify how to make Visual State type definitions.

Choose between:

As typedefs

Uses the `typedef` keyword for type definitions. This is the preferred setting because it helps the compiler to do type checking.

As macros

Uses the `#define` keyword for type definitions. Use this setting if the compiler cannot determine that two type definitions actually are the same.

VS_BOOL type

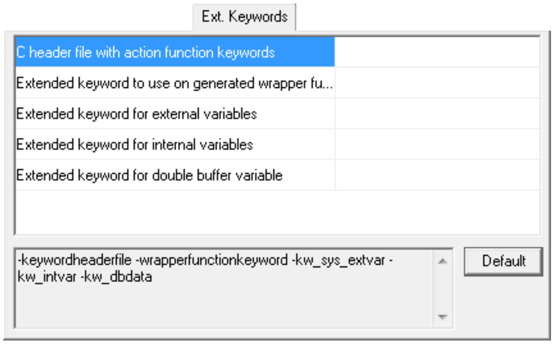
Specify the runtime type to use for the `VS_BOOL` type. By default, the setting is `int`.

Default

Restores the options to their default settings.

Classic Coder Options dialog box : Extended Keywords

The **Extended Keywords** options page contains options for extended keywords.



Use this page to make extended keywords settings for the Classic Coder. The display area under the options shows the resulting command line for the code generation.

This options page can be accessed on both project level and system level. Exactly which options you can set depends on the level you are setting options on.

Extended keyword for system context

Specify an extended keyword string for the system context variables (variable data). This option can only be set on project level.

Extended keyword for external variables

Specify an extended keyword string for external variables (variable data). This option can be set on both project level and system level.

Extended keyword for core model logic

Specify an extended keyword string for the core model logic `struct` variables (constant data). This option can only be set on project level.

Extended keyword for guard expression collection

Specify an extended keyword string for the guard expression collection variables (constant data). This option can only be set on project level.

Extended keyword for action expression collection

Specify an extended keyword string for the action expression collection variables (constant data). This option can only be set on project level.

Extended keyword for runtime info

Specify an extended keyword string for the runtime information `struct` variable (constant data). By default, the runtime information `struct` only contains the digital signature for the project. This option can only be set on project level.

C header file with action function keywords

Specify a C header file that contains keywords for action functions. There is a browse button for your convenience. If an extended keyword is associated with the function prototype either as a keyword or as `#pragma type_attribute`, it will be used during code generation. For a description of the syntax of this file, see *Syntax of C header files*, page 316.

This option can only be set on system level.

An example:

```
#pragma VS_ACTION_BEGIN

#pragma type_attribute=__arm
VS_VOID Action2(VS_INT param1, VS_EVENT_TYPE param2);

__thumb VS_VOID Timer1(VS_UINT event, VS_UINT ticks);

#pragma VS_END
```

Extended keyword to use on generated wrapper functions

Specify an extended keyword to be used for all generated wrapper functions for guards and action calls. This option can only be set on system level.

Extended keyword for internal variables

Specify an extended keyword string for internal variables (variable data). This option can only be set on system level.

Extended keyword for double buffer variable

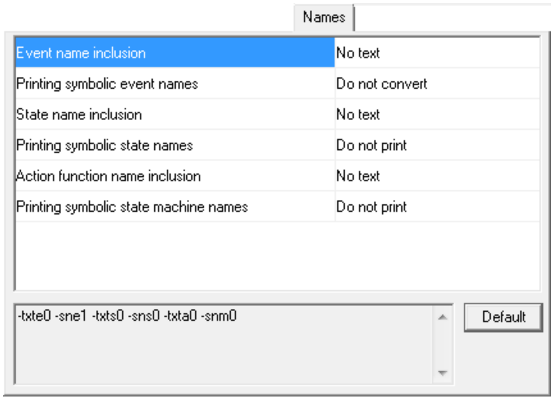
Specify an extended keyword string for the `double` buffer variable (variable data). This option can only be set on system level.

Default

Restores the options to their default settings.

Classic Coder Options dialog box : Names

The **Names** options page contains options for including text associated with states, events, and actions in the generated code.



Use this page to make name settings for the Classic Coder. The display area under the options shows the resulting command line for the code generation.

This options page can only be accessed on system level.

Event name inclusion

Specify the amount of text associated with events to include in the generated code.

Choose between:

No text

Includes no text associated with events in the generated code.

Names included

Includes the names of the events in the generated code. This makes it possible to extract the names from the application when it executes on the target. See the documentation for the API functions with suffixes `_Name` and `_NameAbs`.

Explanations included

Includes the descriptions of the events in the generated code. This makes it possible to extract the descriptions from the application when it executes on the target. See the documentation for the API functions with suffixes `_Expl` and `_ExplAbs`.

Names and explanations

Includes both the names and the descriptions of the events in the generated code.

Printing symbolic event names

Specify how to generate symbolic event names.

Choose between:

Do not print
No symbolic event names are generated.

Do not convert
Generates symbolic event names as defined in the model.

Convert to uppercase
Generates symbolic event names as defined in the model, but converted to upper case.

State name inclusion

Specify the amount of text associated with states to include in the generated code.

Choose between:

No text
Includes no text associated with states in the generated code.

Names included
Includes the names of the states in the generated code. This makes it possible to extract the names from the application when it executes on the target. See the documentation for the API functions with suffixes `_Name` and `_NameAbs`.

Explanations included
Includes the descriptions of the states in the generated code. This makes it possible to extract the descriptions from the application when it executes on the target. See the documentation for the API functions with suffixes `_Expl` and `_ExplAbs`.

Names and explanations
Includes both the names and the descriptions of the states in the generated code.

Printing symbolic state names

Specify how to generate symbolic state names. Typically, this is useful when you use the functions `SEM_State`, `SEM_Machine`, and `SEM_ForceState`.

Choose between:

Do not print
No symbolic state names are generated.

Do not convert

Generates symbolic state names as defined in the model.

Convert to uppercase

Generates symbolic state names as defined in the model, but converted to upper case.

Action function name inclusion

Specify the amount of text associated with action functions to include in the generated code.

Choose between:

No text

Includes no text associated with action functions in the generated code.

Names included

Includes the names of the action functions in the generated code. This makes it possible to extract the names from the application when it executes on the target. See the documentation for the API functions with suffixes `_Name` and `_NameAbs`.

Explanations included

Includes the descriptions of the action functions in the generated code. This makes it possible to extract the descriptions from the application when it executes on the target. See the documentation for the API functions with suffixes `_Expl` and `_ExplAbs`.

Names and explanations

Includes both the names and the descriptions of the action functions in the generated code.

Printing symbolic state machine names

Specify how to generate symbolic state machine names.

Choose between:

Do not print

No symbolic state machine names are generated.

Do not convert

Generates symbolic state machine names as defined in the model.

Convert to uppercase

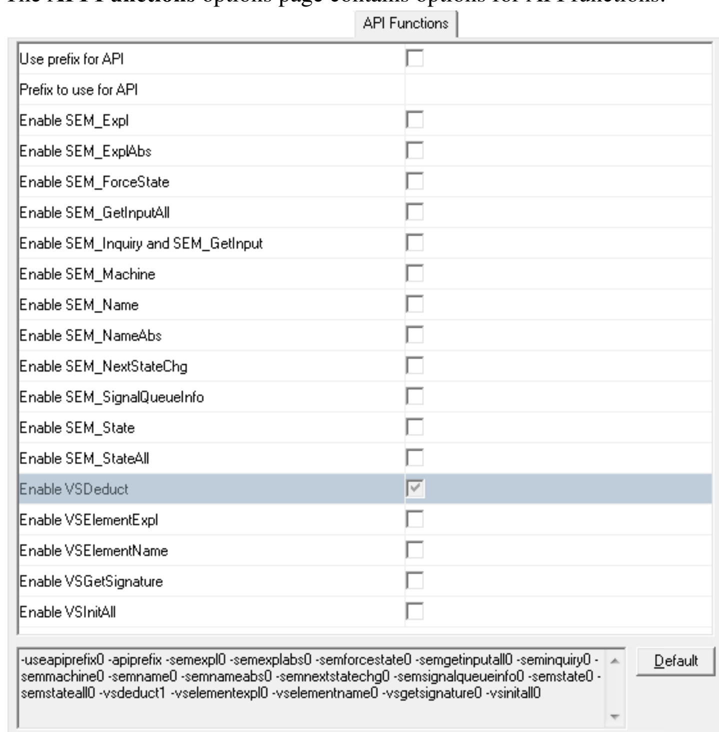
Generates symbolic state machine names as defined in the model, but converted to upper case.

Default

Restores the options to their default settings.

Classic Coder Options dialog box : API Functions

The **API Functions** options page contains options for API functions.



Use this page to make API function settings for the Classic Coder and to enable specific API functions. The display area under the options shows the resulting command line for the code generation.

Note that different sets of options are available if you set the options on system level or on project level. The screenshot reflects the options available on system level.

Use prefix for API

Uses the prefix specified with the **Prefix to use for API** option in front of all identifiers, functions, etc, in the system. This option is only be available on system level.

Prefix to use for API

Specify a prefix to put in front of all identifiers, functions, etc, in the system. This option is only available on system level.

Enable *function*

Enables a specific Adaptive API function. See also *Descriptions of the Adaptive API functions*, page 592.

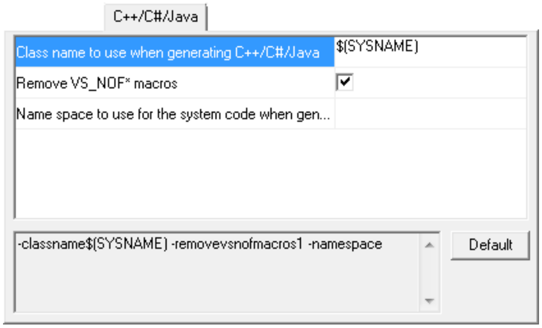
Available functions to enable depends on whether you set the options on system or project level.

Default

Restores the options to their default settings.

Classic Coder Options dialog box : C++/C#/Java

The **C++/C#/Java** options page contains options for C++, C#, and Java code generation.



Use this page to make C++, C#, or Java settings for the Classic Coder. The display area under the options shows the resulting command line for the code generation.

This options page can be accessed on both project level and system level. The screenshot reflects the options available on system level.

See also *Adaptive API table-based code with C++*, page 571.

Class name to use when generating C++/C#/Java

Specify the class name to use for the generated system. It must be a legal identifier. By default, the class name is the name of the system. This option is only available on system level.

Remove VS_NOF* macros

Replaces the `VS_NOF*` macros with methods on the system class. This option is only available on system level.

Name space to use for the project code when generating C++/C#/Java

Specify the C++, C#, or Java namespace for all code related to this project. This option is only available on project level.

Name space to use for the system code when generating C++/C#/Java

Specify the C++, C#, or Java namespace for all code related to this system. This option is only available on system level.

Package name to use for the project code when generating Java

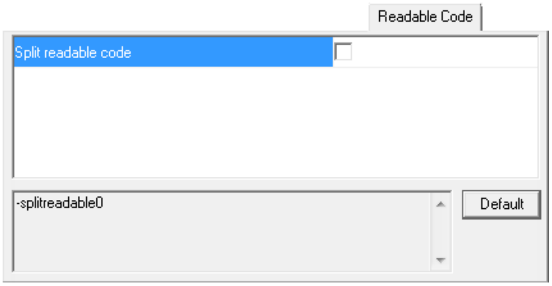
Specify the package name used when generating Java code. It might be a good idea to make sure that the package name is aligned with the output path. This option is available on project level when Java is the selected code output.

Default

Restores the options to their default settings.

Classic Coder Options dialog box : Readable Code

The **Readable Code** options page contains options for generating readable code.



Use this page to make readable code settings for the Classic Coder. The display area under the options shows the resulting command line for the code generation.

This options page can only be accessed on system level.

See also *Adaptive API readable code*, page 571.

Split readable code

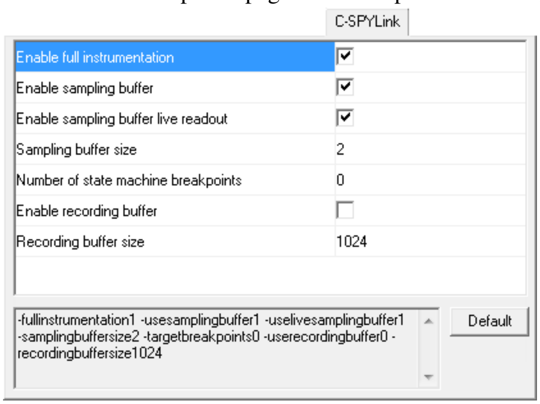
Rewrites all `SystemVSDeduct` functions to use helper functions for event processing. This can be beneficial for very large `SystemVSDeduct` functions, because it reduces the compilation time if aggressive compiler optimizations are used. It can also overcome any arbitrary implementation function size limits of your compiler. This option causes a small increase in code size and a small reduction in execution speed.

Default

Restores the options to their default settings.

Classic Coder Options dialog box : C-SPYLink

The **C-SPYLink** options page contains options for debugging using C-SPYLink.



Use this page to make C-SPYLink settings for the Classic Coder. The display area under the options shows the resulting command line for the code generation.

This options page can be accessed on both project level and system level. Depending on which level you set options on, different sets of options are available.

See also *Debugging design models using C-SPYLink*, page 759.

Enable using shared DLIB breakpoint

Makes the generated code use the shared breakpoint available in the DLIB runtime environment. If the number of breakpoints is limited, this helps to preserve the number of allocated breakpoints. Do not use this option with the legacy CLIB runtime environment. This option can only be set on project level.

Enable using ARM EABI shared semi-hosting breakpoint

Makes the generated code use the shared semi-hosting breakpoint available in the Arm EABI-specific runtime environment. If the number of breakpoints is limited, this helps to preserve the number of allocated breakpoints. This option requires IAR Embedded Workbench® for Arm 5.10 or later. This option can only be set on project level.

Suppress C-SPYLink common files

Prevents multiple C-SPYLink files from being generated when you are using two or more projects in the same linked image together with C-SPYLink. This option can only be set on project level.

Enable full instrumentation

Extracts a maximum amount of debug information from your model. This option causes a small increase in code size and a significant reduction in execution speed. This option can only be set on system level.

Enable sampling buffer

Enables on-target sampling buffers for a single macrostep. C-SPYLink will be able to extract large amounts of debug information from your model. This option causes an increase in code size and a small reduction in execution speed. If sequence recording is used, the speed reduction will be larger. Use the option **Sampling buffer size** to set the size of the buffer.

This option can only be set on system level.

Enable sampling buffer readout

Reads data from the sampling buffer while the target application is executing. The target controller must support live read. This option can only be set on system level.

Sampling buffer size

Set the number of elements in the sampling buffer for C-SPYLink. If the value is too low, you can only see the event that triggered the most recent transition and the states after that microstep. If the value is too high, the target application might run out of memory. This option does not change the behavior of the model.

This option can only be set on system level.

Number of state machine breakpoints

Set the number of available breakpoints for C-SPYLink on the target controller. Using this option consumes memory. This option does not change the behavior of the model.

This option can only be set on system level.

Enable recording buffer

Makes it possible to make recordings (execution logs) at almost full speed. This option also makes it possible to display sampled data back. Use the option **Recording buffer size** to set the size of the buffer.

This option can only be set on system level.

Recording buffer size

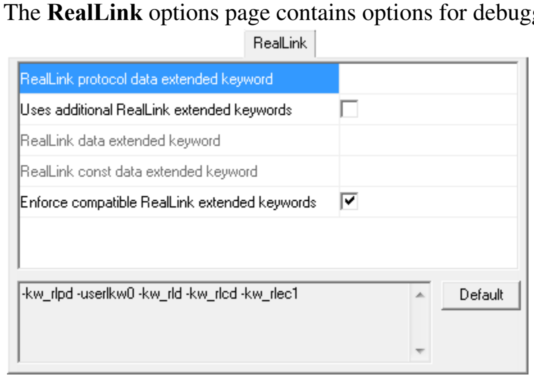
Set the number of elements in the recording buffer for C-SPYLink. This option can only be set on system level.

Default

Restores the options to their default settings.

Classic Coder Options dialog box : RealLink

The **RealLink** options page contains options for debugging using RealLink.



Use this page to make RealLink settings for the Classic Coder. The display area under the options shows the resulting command line for the code generation.

This options page can only be accessed on project level.

See also *Debugging design models using C-SPYLink*, page 759.

RealLink protocol data extended keyword

Specify an extended keyword string to use for RealLink protocol data.

Use additional RealLink extended keywords

Enables additional RealLink extended keywords.

RealLink data extended keyword

Specify an extended keyword string to use for RealLink symbol table data.

RealLink const data extended keyword

Specify an extended keyword string to use for RealLink symbol table constant data.

Enforce compatible RealLink extended keywords

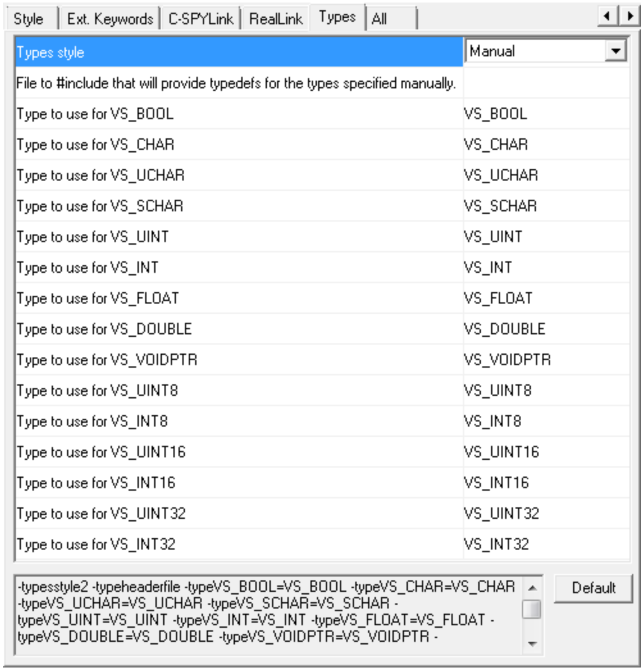
Replaces all standard Visual State extended keywords with RealLink-compatible extended keywords.

Default

Restores the options to their default settings.

Classic Coder Options dialog box : Types

The **Types** options page contains options for data types.



Use this page to specify the underlying data types to be used for Visual State data types. The display area under the options shows the resulting command line for the code generation.

This options page can only be accessed on project level.

See also *Visual State data types*, page 197.

Types style

Selects the underlying data type for the Visual State data types.

Choose between:

VisualState

Uses the standard Visual State types.

C99

Uses C99 data types, where possible, as the underlying types for the generated `VS_*` types.

Manual

Allows you to specify individually for each generated `VS_*` type which underlying data type to use.

File to #include that will provide typedefs for the types specified manually

Specify a file with type definitions to include.

Type to use for *Visual State data type*

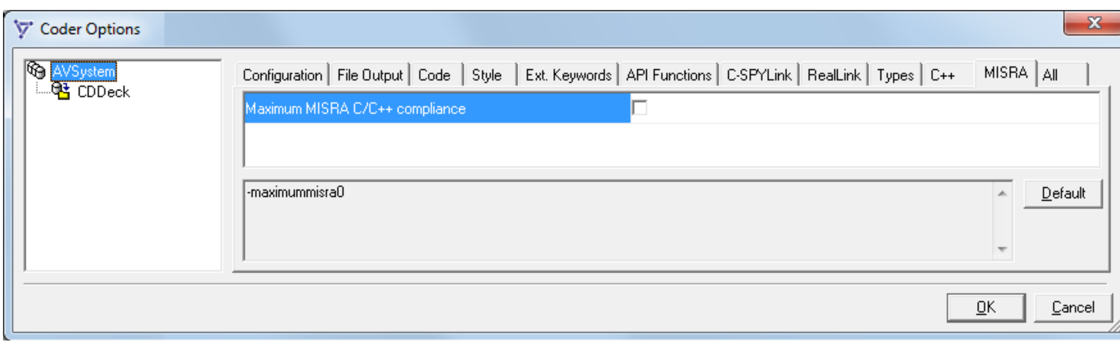
Specify which underlying data type to use for a specific Visual State data type. See also *Visual State data types*, page 197.

Default

Restores the options to their default settings.

Classic Coder Options dialog box : MISRA

The **MISRA** options page contains options for generating code that is more compliant with MISRA C/C++ rules.



Use this page to enable generation of code that is more compliant to MISRA C/C++. The display area under the options shows the resulting command line for the code generation.

This options page can only be accessed on project level.

Maximum MISRA C/C++ compliance

Changes the generated code so that it is more compliant with MISRA C/C++. This causes these other Classic Coder options to be automatically set:

- `-removevsnofmacros1` will be automatically set if you have enabled C++ code generation

- -tsemt0
- -tvst0

These macros will not be used:

- the macros used for getting the signature: VS_RUNTIME_INFO, VS_RUNTIME_INFO_EXTKW, VS_SIGNATURE_VERSION, VS_SIGNATURE_CONTENT, VS_SIGNATURE_VERSION_LENGTH, and VS_SIGNATURE_CONTENT_LENGTH. Make sure to enable and use the function VSGetSignature instead.
- the macros VS_TRUE and VS_FALSE. If you need them for other reasons, add these macros to your state machine model as constants instead.

These types will be changed:

- VS_VOID will not be used by the generated code. Use plain void instead.
- The new enumeration type VSResult will contain the SES_*** members in the project header (and project name space).
- The VS_*** types (for example, VS_UINT8) will be put in the project header file (and project name space).

If you have enabled C++ code generation, these changes also apply:

- the symbolic identifier names (events, states, and action functions) for a system will be generated as enumerations in the system class. Calls that must use these symbolic names must use for example, {*system instance*}/*class name*}.*Event1*.
- these functions will not be generated:
  - getNofActionExpressions
  - getNofActionFunctions
  - getNofEventGroups
  - getNofExternalVariables
  - getNofGuardExpressions
  - getNofInternalVariables
  - getNofSignals
- static cast will be used instead of old style C cast.
- the project files can optionally have their own name space.
- the Visual State constants will be generated as const variables.

SEM type identifiers

The SEM type identifiers are defined in the Classic Coder-generated file `SEMTypes.h`.

These are the available SEM type identifiers:

Type identifiers	Description
SEM_EVENT_TYPE	Event data type
SEM_EVENT_GROUP_TYPE	Reserved for internal use in the Visual State APIs
SEM_GUARD_EXPRESSION_TYPE	Reserved for internal use in the Visual State APIs
SEM_STATE_TYPE	State data type
SEM_ACTION_FUNCTION_TYPE	Action function data type. Used only for action functions without parameters and which have the return type VS_VOID.
SEM_ACTION_EXPRESSION_TYPE	Action expression data type.
SEM_SIGNAL_QUEUE_TYPE	Signal queue data type.
SEM_INSTANCE_TYPE	Instance data type.
SEM_STATE_MACHINE_TYPE	State machine data type.
SEM_EXPLANATION_TYPE	Explanation data type.
SEM_INTERNAL_TYPE	Reserved for internal use in the Visual State APIs.
SEM_RULE_INDEX_TYPE	Reserved for internal use in the Visual State APIs.
SEM_RULE_TABLE_INDEX_TYPE	Reserved for internal use in the Visual State APIs.
SEM_EGTI_TYPE	Reserved for internal use in the Visual State APIs.

Table 31: SEM type identifiers

Transition rule data format

The transition rule data format is used for storing transitions in the local code layer. Each transition rule consists of one rule data header word and one rule data element for each element of the transition rule.

By default, the Classic Coder will optimize the size of the rule data format number.

For projects that do not use guard expressions or signals, you can apply data formats with all data header types (type 1, 2, or 3). For projects that contain guard expressions or signals, you must apply a format with rule data header word type 2 or 3. It is always possible to force the Classic Coder to use a larger format than the format determined by the Coder.

This table shows the rule data header word type, rule data header word width, and the rule data width of the different transition rule data formats:

Rule data format number	Rule data header word type	Rule data header word width	Rule data width
0	Type 1	16 bits	8 bits
1	Type 2	24 bits	8 bits
2	Type 1	32 bits	8 bits
3	Type 2	48 bits	8 bits
4	Type 1	16 bits	16 bits
5	Type 3	32 bits	16 bits
6	Type 1	32 bits	16 bits
7	Type 2	48 bits	16 bits
8	Type 1	32 bits	32 bits
9	Type 3	64 bits	32 bits

Table 32: Transition rule data format

Classic Coder command line options

- Introduction to invoking the Classic Coder using command line options
- Summary of Classic Coder options
- Descriptions of Classic Coder options.

Introduction to invoking the Classic Coder using command line options

Learn more about:

- *Briefly about invoking the Classic Coder*, page 701
- *Invocation syntax for the Classic Coder*, page 702

BRIEFLY ABOUT INVOKING THE CLASSIC CODER

You can set Classic Coder options either in the Navigator—using the **Classic Coder Options** dialog box—or via the command line using command line options.

A Coder option is either a project option or a system option. In general, project options affect the project and all systems part of it. System options only affect the systems for which they are specified.

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 apply to all systems.

Coder options are categorized based on these types:

Enumeration options	[E]
Integral options	[I]
Text options	[T]
Boolean options	[B]

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.

For a complete list of available Classic Coder options, run the `Coder.exe` from the command prompt.

INVOCATION SYNTAX FOR THE CLASSIC CODER

This is the invocation syntax for starting the Classic Coder from the command line:

```
Coder.exe vsp_file [--l] [--@option-file] -option[argument]*
```

Where:

`--l` loads options from the `vtg` file associated with the specified `vsp` file.

`--@`*`option-file`* loads 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 `//`.

Example 1

```
Coder.exe Mobile.vsp
```

Description: Generates code for the project and stores it in the file `Mobile.vsp`.

Example 2

```
Coder.exe Mobile.vsp -api_type1 -Vmobile1 -txte3 -txts3 -txta3
-Vmobile2
```

Description: Generates code for the project, which contains the systems `Mobile1` and `Mobile2`. Code is generated for the Uniform API.

In addition, the system `Mobile1` will be generated with names and descriptions for events, states, and action functions.

Example 3

```
Coder.exe Mobile.vsp --@MobileSetup.txt -Vmobile -txte3 -txts3
-txta3
```

Description: Generates code for the project, which contains the system `Mobile`. Code is generated for the Adaptive API.

In addition, the system `Mobile` will be generated with names and descriptions for events, states, and action functions.

Summary of Classic Coder options

This table summarizes the Classic Coder command line options:

Command line option	Description
`-apiprefix`	Specifies a prefix to put in front of all identifiers, functions, etc, in the system. [System option]
`-api_type`	Specifies the runtime API to use for code generation. [Project option]
`-armsemihostingbreakpoint`	Determines whether the generated code uses the shared Arm EABI semi-hosting breakpoint. [Project option]
`-autoentryfunction`	Adds a call to a predefined function whenever a state is entered. [Project option]
`-autoexitfunction`	Adds a call to a predefined function whenever a state is exited. [Project option]
`-classname`	Specifies the class name to use for the generated system. [System option]
`-constactionfpt`	Determines whether the action expression function pointer table is defined as a `const` variable. [System option]
`-constcml`	Determines whether the core model logic is defined as a `const` variable. [System option]
`-constguardfpt`	Determines whether the guard expression function pointer table is defined as a `const` variable. [System option]
`-cppcode`	Specifies that C++ code will be generated. [Project option]
`-cppsourcefileext`	Determines the filename extension that IAR Visual State uses for generated C++ language source files. [Project option]
`-cscode`	Specifies that C# code will be generated. [Project option]
`-csourcefileext`	Determines the filename extension that IAR Visual State uses for generated C language source files. [Project option]
`-cspylink`	Determines whether the generated code can be debugged using C-SPYLink. [Project option]
`-D`	Specifies the data width for SEM data types for the entire project. [Project option]
`-dlibbreakpoint`	Determines whether the generated code uses the shared DLIB breakpoint. [Project option]
`-dw`	Specifies the data width for SEM data types for a specific system. [System option]

Table 33: Classic Coder command line options

Command line option	Description
-fullinstrumentation	Controls the amount of debug information that C-SPYLink can extract from your model. [System option]
-funcexph	Specifies how the Classic Coder should handle functional expressions. [Project option]
-gds	Determines whether the Classic Coder includes a digital signature in the generated code. [Project option]
-generatetimeandvers ion	Determines whether the Classic Coder includes the time of the code generation and the version of the Coder in the generated code. [Project option]
-H	Specifies the name of the header file that contains system-level model declarations. [System option]
-iev	Specifies how to initialize external variables. [Project option]
-iiv	Specifies how to initialize internal variables. [Project option]
-include_excluded	Determines whether the Classic Coder generates code also for graphical objects marked as excluded in the Designer. [Project option]
-jvcode	Specifies that Java code will be generated. [Project option]
-keywordheaderfile	Specifies a C header file that contains keywords for action functions. [System option]
-kw_actionexpr	Specifies an extended keyword string for the action expression collection variables. [Project option]
-kw_context	Specifies an extended keyword string for the system context variables. [Project option]
-kw_corelogic	Specifies an extended keyword string for the core model logic `struct` variables. [Project option]
-kw_dbdata	Specifies an extended keyword string for the `double` buffer variable. [System option]
-kw_guardexpr	Specifies an extended keyword string for the guard expression collection variables. [Project option]
-kw_intvar	Specifies an extended keyword string for internal variables. [System option]
-kw_prj_extvar	Specifies an extended keyword string for external variables in the entire project. [Project option]
-kw_rlcd	Specifies an extended keyword string used for RealLink symbol table `const` data. [Project option]

Table 33: Classic Coder command line options (Continued)

Command line option	Description
-kw_rld	Specifies an extended keyword string used for RealLink symbol table data. [Project option]
-kw_rlec	Controls whether to replace all standard Visual State extended keywords with RealLink-compatible extended keywords. [Project option]
-kw_rlpd	Specifies an extended keyword string used for RealLink protocol data. [Project option]
-kw_runtimeinfo	Specifies an extended keyword string for the runtime information `struct` variable. [Project option]
-kw_sys_extvar	Specifies an extended keyword string for external variables in a system. [System option]
-namespace	Specifies the C++/C# namespace for all code related to the system. [System option]
-no_warnings	Determines whether warnings should be disabled. [Project option]
-oa	Determines whether the Classic Coder merges action expressions. [System option]
-og	Determines whether the Classic Coder merges guard expressions. [System option]
-omitcontradictiontests	Controls the generation of contradiction test code for each transition. [System option]
-osm	Controls optimization of states and state machines. [Project option]
-path	Specifies the output path for all generated project files. [Project option]
-projectheader	Specifies the name of the header file that contains project-level model declarations. [Project option]
-projectnamespace	Specifies the project name space to use for C++ output for project-related types and functions. [Project option]
-projectpackage	Specifies the package name used when generating Java code. [Project option]
-projectsource	Specifies the name of the source file that contains project-level model definitions. [Project option]
-R	Specifies a name for a report file to contain information about the project. [Project option]

Table 33: Classic Coder command line options (Continued)

Command line option	Description
-rdfm	Specifies the transition rule data format to use for the whole project when generating code. [Project option]
-readable	Determines whether to generate table-based or readable code. [Project option]
-reallink	Determines whether the generated code can be debugged using RealLink. [Project option]
-recordingbuffersize	Specifies the number of elements in the recording buffer for C-SPYLink. [System option]
-removevsnofmacros	Specifies whether the VS_NOF* macros are replaced with methods on the system class. [System option]
-S	Specifies the name of the source file that contains system-level model definitions. [System option]
-samplingbuffersize	Specifies the number of elements in the sampling buffer for C-SPYLink. [System option]
-sem*func*	Specifies whether to enable the API function SEM_*func*. [System option]
-sne	Controls how symbolic event names are generated. [System option]
-snm	Controls how symbolic state machine names are generated. [System option]
-sns	Controls how symbolic state names are generated. [System option]
-spath	Specifies the output path for all generated system files. [System option]
-splitreadable	Specifies whether the Classic Coder rewrites all *systemname*VSDeduct functions to use helper functions for event processing. [System option]
-suppress_cspylink_common_files	Controls how multiple C-SPYLink files are generated when you are using two or more projects in the same linked image. [Project option]
-sysrdfm	Specifies the transition rule data format to use for a specific system when generating code. [System option]
-targetbreakpoints	Specifies the number of available breakpoints for C-SPYLink on the target controller. [System option]
-tsemt	Specifies how to make SEM type definitions. [Project option]

Table 33: Classic Coder command line options (Continued)

Command line option	Description
-tvsvt	Specifies how to make Visual State type definitions. [Project option]
-txta	Controls the amount of text associated with action functions to include in the generated code. [System option]
-txte	Controls the amount of text associated with events to include in the generated code. [System option]
-txts	Controls the amount of text associated with states to include in the generated code. [System option]
-type*VStype*	Specifies which underlying data type to use for the generated Visual State data type. [Project option]
-typeheaderfile	Specifies a header file to include in files that need type definitions to declare manually specified underlying data types for the Visual State data types. [Project option]
-typestyle	Selects the underlying data types for the generated Visual State data types. [Project option]
-useapiprefix	Determines whether to use the prefix specified with the -apiprefix option in front of all identifiers, functions, etc. [System option]
-useautovariables	Determines whether auto variables are allowed in the generated API code. [System option]
-useguardtypecast	Determines whether the Classic Coder uses guard type casts. [System option]
-useheap	Determines whether the Classic Coder uses heap memory. [Project option]
-uselivesamplingbuffer	Determines whether C-SPYLink can read data from the sampling buffer while the target application is executing. [System option]
-usepop	Determines whether the Classic Coder uses the same output path for system files as the path specified for all project files. [System option]
-userecordingbuffer	Determines whether to use a recording buffer. [System option]
-userfileinclusion	Specifies a file to include in every generated source file. [Project option]
-userlkw	Specifies whether to use additional RealLink extended keywords. [Project option]

Table 33: Classic Coder command line options (Continued)

Command line option	Description
-usesamplingbuffer	Controls on-target sampling buffers for a single macrostep. [System option]
-V	Specifies the system that the following system options apply to. [System option]
-variant	Specifies which variant to generate code for. [Project option]
-vsbooltype	Specifies the data type to use for the `VS_BOOL` type at runtime. [Project option]
-vsdeduct	Enables generation of the `VSDeduct` function. [System option]
-vselementexpl	Enables generation of the `VSElementExpl` function. [System option]
-vselementname	Enables generation of the `VSElementName` function. [System option]
-vsinitall	Enables generation of the `VSInitAll` function. [System option]
-vsgetsignature	Enables generation of the `VSGetSignature` function. [Project option]
-warnings_affect_exit_code	Determines whether warnings generate a non-zero exit code. [Project option]
-warnings_are_errors	Determines whether all warnings are reclassified as errors. [Project option]
-wrapperfunctionkeyword	Specifies an extended keyword to be used for all generated wrapper functions for guards and action calls. [System option]

Table 33: Classic Coder command line options (Continued)

Descriptions of Classic Coder options

The following pages give detailed reference information about each Classic Coder command line option.

-apiprefix

Syntax

`-apiprefix`*`prefix`*

Parameters

`prefix` A string that will be used as a prefix for all identifiers, functions, etc, in the system.

Scope — System level.

Description — Specifies a prefix to put in front of all identifiers, functions, etc, in the system. This option requires that you have specified the option `-useapiprefix1`.

See also — *-useapiprefix*, page 747.

Project>Options>Code Generation>*system*>API Functions>Prefix to use for API

-api_type

Syntax — `-api_type{0|1}`

Parameters

`0` (default)	The Adaptive API, which is optimized for the data size of each system and has a copy of the API functions for each system.
`1`	The Uniform API, which uses one shared API for all systems and uses the same data sizes for all systems.

Scope — Project level.

Description — Specifies the runtime API to use for code generation.

See also — *The Visual State APIs*, page 459.

Project>Options>Code Generation>*project*>Configuration>API type

-armsemihostingbreakpoint

Syntax — `-armsemihostingbreakpoint{0|1}`

Parameters

`0` (default)	The generated code does not use the shared semi-hosting breakpoint available in the Arm EABI-specific runtime environment.
`1`	The generated code uses the shared semi-hosting breakpoint available in the Arm EABI-specific runtime environment.

Scope — Project level.

Description — Determines whether the generated code uses the shared semi-hosting breakpoint available in the Arm EABI-specific runtime environment. If the number of breakpoints is limited, using this breakpoint helps to preserve the number of allocated breakpoints. This option requires IAR Embedded Workbench® for Arm 5.10 or later.

See also — *-dlibbreakpoint*, page 716.

Project>Options>Code Generation>*project*>C-SPYLink>Enable using ARM EABI shared semi-hosting breakpoint

-autoentryfunction

Syntax — `-autoentryfunction`*`funcname`*

Parameters

`funcname`	A user-defined function that is called whenever a state is entered.

Scope — Project level.

Description — Specifies a user-defined function to call whenever a state is entered.

Example — `-autoentryfunctionMy_Function`

See also — *VSProjectEnterState*, page 477.

Project>Options>Code Generation>*project*>Code>Automatic entry function call

-autoexitfunction

Syntax — `-autoexitfunction`*`funcname`*

Parameters

`funcname`	A user-defined function that is called whenever a state is exited.

Scope — Project level.

Description — Specifies a user-defined function to call whenever a state is exited.

Example — `-autoexitfunctionMy_Function`

See also — *VSProjectLeaveState*, page 481.

Project>Options>Code Generation>*project*>Code>Automatic exit function call

-classname

Syntax — `-classname`*`identifier`*

Parameters

`identifier`	A string that specifies the class name for the generated system.

Scope — System level.

Description — Specifies the class name to use for the generated system. It must be a legal identifier. By default, the class name is the name of the system.

Project>Options>Code Generation>*system*>C++/C#/Java>Class name to use when generating C++/C#/Java

-constactionfpt

Syntax — `-constactionfpt{0|1}`

Parameters

`0`	The action expression function pointer table is not defined as a `const` variable.
`1` (default)	Defines the action expression function pointer table as a `const` variable.

Scope — System level.

Description — Determines whether the action expression function pointer table is defined as a `const` variable. This option should always be set to `1` except in exceptional cases, for example, when the target controller has sufficient and fast RAM, and speed is of the highest importance.

See also — *-constcml*, page 712 and *-constguardfpt*, page 712.

Project>Options>Code Generation>*system*>Code>Const action expression FPT

-constcml

Syntax `-constcml{0|1}`

Parameters

`0`	The core model logic is not defined as a `const` variable.
`1` (default)	Defines the core model logic as a `const` variable.

Scope System level.

Description Determines whether the core model logic is defined as a `const` variable. This option should always be set to `1` except in exceptional cases, for example, when the target controller has sufficient and fast RAM, and speed is of the highest importance.

See also -*constactionfpt*, page 711 and -*constguardfpt*, page 712.

Project>Options>Code Generation>*system*>Code>Const core model logic

-constguardfpt

Syntax `-constguardfpt{0|1}`

Parameters

`0`	The guard expression function pointer table is not defined as a `const` variable.
`1` (default)	Defines the guard expression function pointer table as a `const` variable.

Scope System level.

Description Determines whether the guard expression function pointer table is defined as a `const` variable. This option should always be set to `1` except in exceptional cases, for example, when the target controller has sufficient and fast RAM, and speed is of the highest importance.

See also -*constactionfpt*, page 711 and -*constcml*, page 712.

Project>Options>Code Generation>*system*>Code>Const guard expression FPT

-cppcode

Syntax `-cppcode{0|1}`

Parameters

`0` (default)	Generates C code unless one of the options `-cscode1` or `-jvcode1` has been specified.
`1`	Generates C++ code.

Scope Project level.

Description Determines whether C++ code will be generated or not. `-cppcode1` cannot be specified together with either `-cscode1` or `-jvcode1`.

See also *Generating code for an API*, page 572.

Project>Options>Code Generation>*project*>Configuration>C++ code generation

-cppsourcefileext

Syntax `-cppsourcefileext`*`extension`*

Parameters

`extension`	The filename extension that IAR Visual State uses for generated C++ language source files.

Scope Project level.

Description Determines the filename extension that IAR Visual State uses for generated C++ language source files. By default, the filename extension is `cpp`.

Project>Options>Code Generation>*project*>Configuration>Source file extension to use for C++ source files

-cscode

Syntax

`-cscode{0|1}`

Parameters

`0` (default)	Generates C code unless one of the options `-cppcode1` or `-jvcode1` has been specified.
`1`	Generates C# code.

Scope

Project level.

Description

Determines whether C# code will be generated or not. `-cscode1` cannot be specified together with either `-cppcode1` or `-jvcode1`.

See also

Generating code for an API, page 572.

Project>Options>Code Generation>*project*>Configuration>C# code generation

-csourcefileext

Syntax

`-csourcefileext`*`extension`*

Parameters

`extension`	The filename extension that IAR Visual State uses for generated C language source files.

Scope

Project level.

Description

Determines the filename extension that IAR Visual State uses for generated C language source files. By default, the filename extension is `c`.

Project>Options>Code Generation>*project*>Configuration>Source file extension to use for C source files

-cspylink

Syntax

`-cspylink{0|1}`

Parameters

`0` (default)	Does not generate code to be debugged using C-SPYLink.

`1`	Generates code to be debugged using C-SPYLink.

Scope

Project level.

Description

Determines whether the generated code can be debugged using C-SPYLink.

See also

Debugging design models using C-SPYLink, page 759 and *-fullinstrumentation*, page 717.

Project>Options>Code Generation>*project*>Configuration>Generate for C-SPYLink

-D

Syntax

`-D{O|0|1|2}`

Parameters

`O` (default)	Uses the most optimal data widths for SEM type definitions. The width is the smallest possible to reduce the use of variable and constant data.
`0`	Sets the data width of all SEM types to 8 bits. If the target microcontroller handles 8-bit accesses well, this setting probably increases the execution speed.
`1`	Sets the data width of all SEM types to 16 bits. If the target microcontroller handles 16-bit accesses well, this setting probably increases the execution speed.
`2`	Sets the data width of all SEM types to 32 bits. If the target microcontroller handles 32-bit accesses well, this setting probably increases the execution speed.

Scope

Project level.

Description

Specifies the data width for SEM data types.

See also

SEM type identifiers, page 699 and *-dw*, page 716.

This option is not available in the graphical interface.

-dlibbreakpoint

Syntax

`-dlibbreakpoint{0|1}`

Parameters

`0` (default)	The generated code does not use the shared breakpoint available in the DLIB runtime environment.
`1`	The generated code uses the shared breakpoint available in the DLIB runtime environment.

Scope

Project level.

Description

Determines whether the generated code uses the shared breakpoint available in the DLIB runtime environment. If the number of breakpoints is limited, using this breakpoint helps to preserve the number of allocated breakpoints. Do not use this option with the legacy CLIB runtime environment.

See also

-armsemihostingbreakpoint, page 709.

Project>Options>Code Generation>*project*>C-SPYLink>Enable using shared DLIB breakpoints

-dw

Syntax

`-dw{0|1|2}`

Parameters

`0` (default)	Uses the most optimal data widths for SEM type definitions. The width is the smallest possible to reduce the use of variable and constant data.
`0`	Sets the data width of all SEM types to 8 bits. If the target microcontroller handles 8-bit accesses well, this setting probably increases the execution speed.
`1`	Sets the data width of all SEM types to 16 bits. If the target microcontroller handles 16-bit accesses well, this setting probably increases the execution speed.
`2`	Sets the data width of all SEM types to 32 bits. If the target microcontroller handles 32-bit accesses well, this setting probably increases the execution speed.

Scope

System level.

Description
Specifies the data width for SEM data types.

See also
SEM type identifiers, page 699 and *-D*, page 715.

This option is not available in the graphical interface.

-fullinstrumentation

Syntax
`-fullinstrumentation{0|1}`

Parameters

`0` (default)	Disables full instrumentation.
`1`	Enables full instrumentation, to extract a maximum amount of debug information.

Scope
System level.

Description
Controls the amount of debug information that C-SPYLink can extract from your model. Specifying `-fullinstrumentation1` causes a small increase in code size and a significant reduction in execution speed.

Project>Options>Code Generation>*system*>C-SPYLink>Enable full instrumentation

-funcexph

Syntax
`-funcexph{0|1|2}`

Parameters

`0` (default)	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.
`1`	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 alternative settings efficiently.

`2`	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 can convert the switch-case construct into a jump table, this might be the most efficient setting.

Scope

Project level.

Description

Specifies how the Classic Coder should handle functional expressions (guard expressions and action expressions).

Project>Options>Code Generation>*project*>Code>Functional expression handling

-gds

Syntax

`-gds{0|1}`

Parameters

`0` (default)	Does not include a digital signature in the generated code.
`1`	Includes a digital signature in the generated code.

Scope

Project level.

Description

Determines whether the Classic Coder includes a digital signature in the generated code.

See also

Digital signatures for tracking inconsistencies, page 74.

Project>Options>Code Generation>*project*>Code>Generate digital signature

-generatetimeandversion

Syntax

`-generatetimeandversion{0|1}`

Parameters

`0` (default)	Does not include the time and the version in the generated code.
`1`	Includes the time and the version in the generated code.

Scope

Project level.

Description

Determines whether the Classic Coder includes the time of the code generation and the version of the Coder in the generated code.

Project>Options>Code Generation>*project*>Code>Generate time and version

-H

Syntax

`-H`*`file`*

Parameters

file	The name of the header file.

Scope

System level.

Description

Specifies the name of the header file that contains system-level model declarations. The name used by default is *System*.h.

Project>Options>Code Generation>*system*>File Output>System header file

-iev

Syntax

`-iev{1|2}`

Parameters

1	Initializes external variables in a function. Specify this parameter if variables must be reinitialized at some point during execution.
2 (default)	Initializes external variables along with their definition. Specify this parameter if variables only need to be initialized once.

Scope

Project level.

Description

Specifies how to initialize external variables.

Project>Options>Code Generation>*project*>Code>External variable initialization

-iiv

Syntax

`-iiv{1|2}`

Parameters

`1`	Initializes internal variables in a function. Specify this parameter if variables must be reinitialized at some point during execution.
`2` (default)	Initializes internal variables along with their definition. Specify this parameter if variables only need to be initialized once.

Scope

Project level.

Description

Specifies how to initialize internal variables.

Project>Options>Code Generation>*project*>Code>Internal variable initialization

-include_excluded

Syntax

`-include_excluded{0|1}`

Parameters

`0` (default)	No code is generated for graphical objects marked as excluded in the Designer.
`1`	Code is generated also for graphical objects marked as excluded in the Designer.

Scope

Project level.

Description

Determines whether the Classic Coder generates code also for graphical objects marked as excluded in the Designer.

Project>Options>Code Generation>*project*>Configuration>Include excluded items

-jvcode

Syntax `-jvcode{0|1}`

Parameters

`0` (default)	Generates C code unless one of the options `-cppcode1` or `-cscode1` has been specified.
`1`	Generates Java code.

Scope Project level.

Description Determines whether Java code will be generated or not. `-jvcode1` cannot be specified together with either `-cppcode1` or `-cscode1`.

See also *Generating code for an API*, page 572.

Project>Options>Code Generation>*project*>Configuration>Java code generation

-keywordheaderfile

Syntax `-keywordheaderfile`*`path`*

Parameters

`path`	The file path to the header file.

Scope System level.

Description Specifies a C header file that contains keywords for action functions. If an extended keyword is associated with the function prototype either as a keyword or as `#pragma type_attribute`, it will be used during code generation. For a description of the syntax of this file, see *Syntax of C header files*, page 316.

Project>Options>Code Generation>*system*>Ext. Keywords>C header file with action function keywords

-kw_actionexpr

Syntax `-kw_actionexpr`*`keyword`*

Parameters

`keyword`	A string that will be used as a keyword.

Scope Project level.

Description Specifies an extended keyword string for the action expression collection variables (constant data).

Project>Options>Code Generation>*project*>Ext. Keywords>Extended keyword for action expression collection

-kw_context

Syntax `-kw_context`*`keyword`*

Parameters

`keyword`	A string that will be used as a keyword.

Scope Project level.

Description Specifies an extended keyword string for the system context variables (variable data).

Project>Options>Code Generation>*project*>Ext. Keywords>Extended keyword for system context

-kw_corelogic

Syntax `-kw_corelogic`*`keyword`*

Parameters

`keyword`	A string that will be used as a keyword.

Scope Project level.

Description Specifies an extended keyword string for the core model logic `struct` variables (constant data).

Project>Options>Code Generation>*project*>Ext. Keywords>Extended keyword for core model logic

-kw_dbdata

Syntax `-kw_dbdata`*`keyword`*

Parameters

keyword	A string that will be used as a keyword.

Scope System level.

Description Specifies an extended keyword string for the double buffer variable (variable data).

Project>Options>Code Generation>*system*>Ext. Keywords>Extended keyword for double buffer variable

-kw_guardexpr

Syntax `-kw_guardexpr`*`keyword`*

Parameters

keyword	A string that will be used as a keyword.

Scope Project level.

Description Specifies an extended keyword string for the guard expression collection variables (constant data).

Project>Options>Code Generation>*project*>Ext. Keywords>Extended keyword for guard expression collection

-kw_intvar

Syntax `-kw_intvar`*`keyword`*

Parameters

keyword	A string that will be used as a keyword.

Scope System level.

Description | Specifies an extended keyword string for internal variables (variable data).

Project>Options>Code Generation>*system*>Ext. Keywords>Extended keyword for internal variables

-kw_prj_extvar

Syntax `-kw_prj_extvar`*`keyword`*

Parameters

keyword	A string that will be used as a keyword.

Scope Project level.

Description Specifies an extended keyword string for external variables (variable data) in the entire project.

See also -*kw_sys_extvar*, page 726.

Project>Options>Code Generation>*project*>Ext. Keywords>Extended keyword for external variables

-kw_rlcd

Syntax `-kw_rlcd`*`keyword`*

Parameters

keyword	A string that will be used as a keyword.

Scope Project level.

Description Specifies an extended keyword string used for RealLink symbol table `const` data.

See also *Debugging design models using RealLink*, page 785.

Project>Options>Code Generation>*project*>RealLink>RealLink const data extended keyword

-kw_rld

Syntax `-kw_rld`*`keyword`*

Parameters

`keyword`	A string that will be used as a keyword.

Scope Project level.

Description Specifies an extended keyword string used for RealLink symbol table data.

See also *Debugging design models using RealLink*, page 785.

Project>Options>Code Generation>*project*>RealLink>RealLink data extended keyword

-kw_rlec

Syntax `-kw_rlec{0|1}`

Parameters

`0` (default)	Uses standard Visual State extended keywords instead of RealLink-compatible extended keywords.
`1`	Replaces all standard Visual State extended keywords with RealLink-compatible extended keywords.

Scope Project level.

Description Controls whether to replace all standard Visual State extended keywords with RealLink-compatible extended keywords.

See also *Debugging design models using RealLink*, page 785.

Project>Options>Code Generation>*project*>RealLink>Enforce compatible RealLink extended keywords

-kw_rlpd

Syntax `-kw_rlpd`*`keyword`*

Parameters

`keyword`	A string that will be used as a keyword.

Scope Project level.

Description Specifies an extended keyword string used for RealLink protocol data.

See also *Debugging design models using RealLink*, page 785.

Project>Options>Code Generation>*project*>RealLink>RealLink protocol data extended keyword

-kw_runtimeinfo

Syntax `-kw_runtimeinfo`*`keyword`*

Parameters

`keyword`	A string that will be used as a keyword.

Scope Project level.

Description Specifies an extended keyword string for the runtime information `struct` variable (constant data). By default, the runtime information `struct` only contains the digital signature for the project.

Project>Options>Code Generation>*project*>Ext. Keywords>Extended keyword for runtime info

-kw_sys_extvar

Syntax `-kw_sys_extvar`*`keyword`*

Parameters

`keyword`	A string that will be used as a keyword.

Scope System level.

Description Specifies an extended keyword string for external variables (variable data) in a system.

See also — -*kw_prj_extvar*, page 724.

Project>Options>Code Generation>*system*>Ext. Keywords>Extended keyword for external variables

-namespace

Syntax — `-namespacename`

Parameters

name	A string that specifies the namespace for C++ or C# code.

Scope — System level.

Description — Specifies the namespace for all C++/C# code related to the system.

Project>Options>Code Generation>*system*>C++/C#/Java>Name space to use for the system code when generating C++/C#

-no_warnings

Syntax — `-no_warnings{0|1}`

Parameters

`0` (default)	Warnings are issued.
`1`	Warnings are disabled and cannot affect the exit code.

Scope — Project level.

Description — Determines whether warnings should be disabled.

See also — -*warnings_are_errors*, page 755

Project>Options>Code Generation>*project*>Configuration>Ignore warnings

-oa

Syntax `-oa{0|1}`

Parameters

`0` (default)	The Classic Coder does not merge action expressions.
`1`	The Classic Coder merges action expressions.

Scope System level.

Description Determines whether the Classic Coder merges action expressions. This might increase execution speed because multiple action expressions associated with a single transition are generated as a compound statement in the code. The drawback is that one and the same action expression might be generated multiple times if constructs such as entry reactions, exit reactions, or history states are used. Setting this option to `0` might generate smaller code.

See also -*og*, page 728

Project>Options>Code Generation>*system*>Code>Merge action expressions

-og

Syntax `-og{0|1}`

Parameters

`0` (default)	The Classic Coder does not merge guard expressions.
`1`	The Classic Coder merges guard expressions.

Scope System level.

Description Determines whether the Classic Coder merges guard expressions. This might increase execution speed because multiple guard expressions associated with a single transition are generated as a compound statement in the code. The drawback is that one and the same guard expression might be generated multiple times if constructs such as entry reactions, exit reactions, or history states are used. Setting this option to `0` might generate smaller code.

See also -*oa*, page 728

Project>Options>Code Generation>*system*>Code>Merge guard expressions

-omitcontradictiontests

Syntax `-omitcontradictiontests{0|1}`

Parameters

`0` (default)	Disables generation of contradiction test code for each transition.
`1`	Generates contradiction test code for each transition. You should only specify the parameter `1` if you know that your system is free from transition conflicts or if you have particular testing requirements, for example, various branch coverage metrics.

Scope System level.

Description Controls the generation of contradiction test code for each transition. 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 readable code and table-based code.

See also *Briefly about Adaptive API code generation*, page 569.

Project>Options>Code Generation>*system*>Code>Omit contradiction tests

-osm

Syntax `-osm{0|1}`

Parameters

`0`	Does not optimize states and state machines.
`1` (default)	Optimizes states and state machines.

Scope Project level.

Description

Controls optimization of states and state machines. Any state machine with a single state is optimized away to reduce the number of states, state machines, and the size of the core model logic.

Project>Options>Code Generation>*project*>Code>Optimize states and state machines

-path

Syntax

`-pathdirectory`

Parameters

directory	The output path for all generated project files.

Scope

Project level.

Description

Specifies the output path for all generated project files. If the path does not exist, it is created. The path can be relative. By default, generated project files are created in the `coder` directory.

See also

-spath, page 738.

Project>Options>Code Generation>*project*>File Output>Output path

-projectheader

Syntax

`-projectheaderfile`

Parameters

file	The name of the project header file.

Scope

Project level.

Description

Specifies the name of the header file that contains macros, types, and function prototypes meant for the project. The name used by default is `Project.h`.

Project>Options>Code Generation>*project*>File Output>Project header file

-projectnamespace

Syntax `-projectnamespace`*`namespace`*

Parameters

`namespace`	The namespace to use for project types and functions when generating C++ or C# output.

Scope Project level.

Description Specifies the namespace to use for C++ or C# output related to the project.

Project>Options>Code Generation>*project*>C++/C#/Java>Name space to use for the project when generating C++/C#

-projectpackage

Syntax `-projectpackage`*`name`*

Parameters

`name`	The package name used when generating Java code.

Scope Project level.

Description Specifies the package name used for Java output in all files generated for this project. It might be a good idea to make sure that the package name is aligned with the output path.

Project>Options>Code Generation>*project*>C++/C#/Java>Package name to use for the project code when generating Java

-projectsource

Syntax `-projectsource`*`file`*

Parameters

`file`	The name of the project source file.

Scope Project level.

Description Specifies the name of the source code file that contains code meant for the project. The name used by default is `Project.c`.

Project>Options>Code Generation>*project*>File Output>Project source file

-R

Syntax

`-Rfile`

Parameters

file	The name of the report file.

Scope

Project level.

Description

Specifies a name for a report file to contain information about the project, option settings, model characteristics, statistics, and a summary of the code generation.

Project>Options>Code Generation>*project*>File Output>Report file

-rdfm

Syntax

`-rdfm{O|0|1|2|3|4|5|6|7|8|9}`

Parameters

`O` (default)	Uses the most optimal transition rule data format. The Classic Coder determines the optimal rule data format with regard to minimal usage of constant data (size optimization).
`0`	Uses transition rule data format 0. This format uses 8-bit access to rule data and supports transition rules with a maximum of 15 8-bit elements of each type, but does not support guard expressions and signals.
`1`	Uses transition rule data format 1. This format uses 8-bit access to rule data and supports transition rules with a maximum of 15 8-bit elements of each type. It supports guard expressions and signals.
`2`	Uses transition rule data format 2. This format uses 8-bit access to rule data and supports transition rules with a maximum of 255 8-bit elements of each type, but does not support guard expressions and signals.
`3`	Uses transition rule data format 3. This format uses 8-bit access to rule data and supports transition rules with a maximum of 255 8-bit elements of each type. It supports guard expressions and signals.

`4`	Uses transition rule data format 4. This format uses 16-bit access to rule data and supports transition rules with a maximum of 15 16-bit elements of each type, but does not support guard expressions and signals.
`5`	Uses transition rule data format 5. This format uses 16-bit access to rule data and supports transition rules with a maximum of 15 16-bit elements of each type. It supports guard expressions and signals.
`6`	Uses transition rule data format 6. This format uses 16-bit access to rule data and supports transition rules with a maximum of 255 16-bit elements of each type, but does not support guard expressions and signals.
`7`	Uses transition rule data format 7. This format uses 16-bit access to rule data and supports transition rules with a maximum of 255 16-bit elements of each type. It supports guard expressions and signals.
`8`	Uses transition rule data format 8. This format uses 32-bit access to rule data and supports transition rules with a maximum of 255 32-bit elements of each type, but does not support guard expressions and signals.
`9`	Uses transition rule data format 9. This format uses 32-bit access to rule data and supports transition rules with a maximum of 255 32-bit elements of each type. It supports guard expressions and signals.

Scope

Project level.

Description

Specifies the transition rule data format to use for the whole project when generating code.

See also

Transition rule data format, page 699 and -*sysrdfm*, page 740.

This option is not available in the graphical interface.

-readable

Syntax

`-readable{0|1}`

Parameters

`0` (default)	Generates table-based code.
`1`	Generates readable code.

Scope

Project level.

Description Determines whether to generate table-based or readable code.

See also *Briefly about Adaptive API code generation*, page 569.

Project>Options>Code Generation>*project*>Configuration>Readable code generation

-reallink

Syntax `-reallink{0|1}`

Parameters

`0` (default)	Does not generate code to be debugged using RealLink.
`1`	Generates code to be debugged using RealLink.

Scope Project level.

Description Determines whether the generated code can be debugged using RealLink.

See also *Debugging design models using RealLink*, page 785.

Project>Options>Code Generation>*project*>Configuration>Generate code for RealLink

-recordingbuffersize

Syntax `-recordingbuffersize`*`size`*

Parameters

`size`	The number of elements in the recording buffer.

Scope System level.

Description Specifies the number of elements in the recording buffer for C-SPYLink.

See also *-userecordingbuffer*, page 749.

Project>Options>Code Generation>*system*>C-SPYLink>Recording buffer size

-removevsnofmacros

Syntax `-removevsnofmacros{0|1}`

Parameters

`0`	The `VS_NOF*` macros are used.
`1` (default)	The `VS_NOF*` macros are replaced by methods on the system class.

Scope System level.

Description Specifies whether the `VS_NOF*` macros are replaced with methods on the system class.

Project>Options>Code Generation>*system*>C++/C#/Java>Remove VS_NOF* macros

-S

Syntax `-S`*`file`*

Parameters

`file`	The name of the source file.

Scope System level.

Description Specifies the name of the source file that contains system-level model definitions. The name used by default is *`System`*`.c`.

Project>Options>Code Generation>*system*>File Output>System source file

-samplingbuffersize

Syntax `-samplingbuffersize`*`size`*

Parameters

`size`	The number of elements in the sampling buffer.

Scope System level.

Description Specifies the number of elements in the sampling buffer for C-SPYLink. If the value is too low, you can only see the event that triggered the most recent transition and the states after that microstep. If the value is too high, the target application might run out of memory. This option does not change the behavior of the model.

See also

-usesamplingbuffer, page 751.

Project>Options>Code Generation>*system*>C-SPYLink>Sampling buffer size

-sem*func*

Syntax

`-semfunc{0|1}`

Parameters

func The unique part of the name of the function to enable or disable. It can be one of:

- `expl` — specifies the Adaptive API function `SEM_Expl`
- `explabs` — specifies the Adaptive API function `SEM_ExplAbs`
- `forcestate` — specifies the Adaptive API function `SEM_ForceState`
- `getinputall` — specifies the Adaptive API function `SEM_GetInputAll`
- `inquiry` — specifies the Adaptive API functions `SEM_Inquiry` and `SEM_GetInput`
- `machine` — specifies the Adaptive API function `SEM_Machine`
- `name` — specifies the Adaptive API function `SEM_Name`
- `nameabs` — specifies the Adaptive API function `SEM_NameAbs`
- `nextstatechg` — specifies the Adaptive API function `SEM_NextStateChg`
- `signalqueueinfo` — specifies the Adaptive API function `SEM_SignalQueueInfo`
- `state` — specifies the Adaptive API function `SEM_State`
- `stateall` — specifies the Adaptive API function `SEM_StateAll`

`0` Disables the Adaptive API function `SEM_func`.

`1` Enables the Adaptive API function `SEM_func`. This is the default value.

Scope

System level.

Description

Specifies whether to enable the API function `SEM_func`.

Example

To enable the Adaptive API function `SEM_ExplAbs`, specify:

`-semexplabs1`

See also

Descriptions of the Adaptive API functions, page 592.

Project>Options>Code Generation>*system*>API Functions>Enable SEM_*

-sne

Syntax

`-sne{0|1|2}`

Parameters

`0`	No symbolic event names are generated.
`1` (default)	Generates symbolic event names as defined in the model.
`2`	Generates symbolic event names as defined in the model, but converted to upper case.

Scope

System level.

Description

Controls how symbolic event names are generated.

See also

-txte, page 744.

Project>Options>Code Generation>*system*>Names>Printing symbolic event names

-snm

Syntax

`-snm{0|1|2}`

Parameters

`0`	No symbolic state machine names are generated.
`1` (default)	Generates symbolic state machine names as defined in the model.
`2`	Generates symbolic state machine names as defined in the model, but converted to upper case.

Scope

System level.

Description

Controls how symbolic state machine names are generated. Typically, this is useful when you use the functions `SEM_State`, `SEM_Machine`, and `SEM_ForceState`.

Project>Options>Code Generation>*system*>Names>Printing symbolic state machine names

-sns

Syntax

`-sns{0|1|2}`

Parameters

`0`	No symbolic state names are generated.
`1` (default)	Generates symbolic state names as defined in the model.
`2`	Generates symbolic state names as defined in the model, but converted to upper case.

Scope

System level.

Description

Controls how symbolic state names are generated.

See also

-*txts*, page 744.

Project>Options>Code Generation>*system*>Names>Printing symbolic state names

-spath

Syntax

`-spath`*`directory`*

Parameters

`directory`	The output path for all generated system files.

Scope

System level.

Description

Specifies the output path for all generated system files. If the path does not exist, it is created. The path can be relative. By default, generated system files are created in the `coder` directory.

See also

-*path*, page 730.

Project>Options>Code Generation>*system*>File Output>Output path

-splitreadable

Syntax `-splitreadable{0|1}`

Parameters

`0` (default)	The Classic Coder does not rewrite any `SystemVSDeduct` functions to use helper functions for event processing.
`1`	The Classic Coder rewrites all `SystemVSDeduct` functions to use helper functions for event processing.

Scope System level.

Description Specifies whether the Classic Coder rewrites all `SystemVSDeduct` functions to use helper functions for event processing. This can be beneficial for very large `SystemVSDeduct` functions, because it reduces the compilation time if aggressive compiler optimizations are used. It can also overcome any arbitrary implementation function size limits of your compiler. This option causes a small increase in code size and a small reduction in execution speed.

See also *Size of generated readable code*, page 462.

Project>Options>Code Generation>*system*>Readable Code>Split readable code

-suppress_cspylink_common_files

Syntax `-suppress_cspylink_common_files{0|1}`

Parameters

`0` (default)	Disables generation of multiple C-SPYLink files when you are using two or more projects in the same linked image together with C-SPYLink.
`1`	Generates multiple C-SPYLink files when you are using two or more projects in the same linked image together with C-SPYLink.

Scope Project level.

Description

Controls how multiple C-SPYLink files are generated when you are using two or more projects in the same linked image together with C-SPYLink.

Project>Options>Code Generation>*project*>C-SPYLink>Suppress C-SPYLink common files

-sysrdfm

Syntax

`-sysrdfm{O|0|1|2|3|4|5|6|7|8|9}`

Parameters

`O` (default) Uses the most optimal transition rule data format. The Classic Coder determines the optimal rule data format with regard to minimal usage of constant data (size optimization).

`0` Uses transition rule data format 0. This format uses 8-bit access to rule data and supports transition rules with a maximum of 15 8-bit elements of each type, but does not support guard expressions and signals.

`1` Uses transition rule data format 1. This format uses 8-bit access to rule data and supports transition rules with a maximum of 15 8-bit elements of each type. It supports guard expressions and signals.

`2` Uses transition rule data format 2. This format uses 8-bit access to rule data and supports transition rules with a maximum of 255 8-bit elements of each type, but does not support guard expressions and signals.

`3` Uses transition rule data format 3. This format uses 8-bit access to rule data and supports transition rules with a maximum of 255 8-bit elements of each type. It supports guard expressions and signals.

`4` Uses transition rule data format 4. This format uses 16-bit access to rule data and supports transition rules with a maximum of 15 16-bit elements of each type, but does not support guard expressions and signals.

`5` Uses transition rule data format 5. This format uses 16-bit access to rule data and supports transition rules with a maximum of 15 16-bit elements of each type. It supports guard expressions and signals.

`6` Uses transition rule data format 6. This format uses 16-bit access to rule data and supports transition rules with a maximum of 255 16-bit elements of each type, but does not support guard expressions and signals.

`7` Uses transition rule data format 7. This format uses 16-bit access to rule data and supports transition rules with a maximum of 255 16-bit elements of each type. It supports guard expressions and signals.

`8`	Uses transition rule data format 8. This format uses 32-bit access to rule data and supports transition rules with a maximum of 255 32-bit elements of each type, but does not support guard expressions and signals.
`9`	Uses transition rule data format 9. This format uses 32-bit access to rule data and supports transition rules with a maximum of 255 32-bit elements of each type. It supports guard expressions and signals.

Scope

System level.

Description

Specifies the transition rule data format to use for a specific system when generating code.

See also

Transition rule data format, page 699 and *-rdfm*, page 732.

This option is not available in the graphical interface.

-targetbreakpoints

Syntax

`-targetbreakpoints`*`number`*

Parameters

`number`	The number of available breakpoints.

Scope

System level.

Description

Specifies the number of available breakpoints for C-SPYLink on the target controller. Target breakpoints speed up execution but consume memory. This option does not change the behavior of the model.

Project>Options>Code Generation>*system*>C-SPYLink>Number of state machine breakpoints

-translatecomments

Syntax

`-translatecomments{0|1}`

Parameters

`0` (default)	Comments in the generated code are not translated to the language that IAR Visual State is configured to run in.

`1`	Comments in the generated code are translated from English to the language that IAR Visual State is configured to run in.

Scope

Project level.

Description

Specifies whether to translate the comments in the generated code to another language than English.

Project>Options>Code Generation>*project*>Code>Translate comments

-tsemt

Syntax

`-tsemt{0|1}`

Parameters

`0` (default)	Uses the `typedef` keyword for type definitions. Specify this value if possible, because it helps the compiler to do type checking.
`1`	Uses the `#define` keyword for type definitions. This value must be specified if the compiler cannot determine that two type definitions actually are the same.

Scope

Project level.

Description

Specifies how to make SEM type definitions.

See also

SEM type identifiers, page 699.

Project>Options>Code Generation>*project*>Style>SEM type definitions

-tvsvt

Syntax

`-tvsvt{0|1}`

Parameters

`0` (default)	Uses the `typedef` keyword for type definitions. Specify this value if possible, because it helps the compiler to do type checking.

`1`	Uses the `#define` keyword for type definitions. This value must be specified if the compiler cannot determine that two type definitions actually are the same.

Scope

Project level.

Description

Specifies how to make Visual State type definitions.

Project>Options>Code Generation>*project*>Style>VS type definitions

-txta

Syntax

`-txta{0|1|2|3}`

Parameters

`0` (default)	Includes no text associated with action functions in the generated code.
`1`	Includes the names of the action functions in the generated code. This makes it possible to extract the names from the application when it executes on the target. See the documentation for the API functions with suffixes `_Name` and `_NameAbs`.
`2`	Includes the descriptions of the action functions in the generated code. This makes it possible to extract the descriptions from the application when it executes on the target. See the documentation for the API functions with suffixes `_Expl` and `_ExplAbs`.
`3`	Includes both the names and the descriptions of the action functions in the generated code.

Scope

System level.

Description

Controls the amount of text associated with action functions to include in the generated code.

Project>Options>Code Generation>*system*>Names>Action function name inclusion

-txte

Syntax
`-txte{0|1|2|3}`

Parameters

`0` (default)	Includes no text associated with events in the generated code.
`1`	Includes the names of the events in the generated code. This makes it possible to extract the names from the application when it executes on the target. See the documentation for the API functions with suffixes `_Name` and `_NameAbs`.
`2`	Includes the descriptions of the events in the generated code. This makes it possible to extract the descriptions from the application when it executes on the target. See the documentation for the API functions with suffixes `_Expl` and `_ExplAbs`.
`3`	Includes both the names and the descriptions of the events in the generated code.

Scope
System level.

Description
Controls the amount of text associated with events to include in the generated code.

See also
-*sne*, page 737.

Project>Options>Code Generation>*system*>Names>Event name inclusion

-txts

Syntax
`-txts{0|1|2|3}`

Parameters

`0` (default)	Includes no text associated with states in the generated code.
`1`	Includes the names of the states in the generated code. This makes it possible to extract the names from the application when it executes on the target. See the documentation for the API functions with suffixes `_Name` and `_NameAbs`.

`2`	Includes the descriptions of the states in the generated code. This makes it possible to extract the descriptions from the application when it executes on the target. See the documentation for the API functions with suffixes `_Expl` and `_ExplAbs`.
`3`	Includes both the names and the descriptions of the states in the generated code.

Scope

System level.

Description

Controls the amount of text associated with states to include in the generated code.

See also

-*sns*, page 738.

Project>Options>Code Generation>*system*>Names>State name inclusion

-type*VStype*

Syntax

`-type`*`VStypeUnderlyingtype`*

Parameters

`VStype`	The generated Visual State data type. It can be one of:
	`VS_BOOL`
	`VS_CHAR`
	`VS_UCHAR`
	`VS_SCHAR`
	`VS_UINT`
	`VS_INT`
	`VS_FLOAT`
	`VS_DOUBLE`
	`VS_VOIDPTR`
	`VS_UINT8`
	`VS_INT8`
	`VS_UINT16`
	`VS_INT16`
	`VS_UINT32`
	`VS_INT32`

`Underlyingtype`	A text string that specifies the underlying data type to use for the generated Visual State data type *`VStype`*. It must be a legal data type.

Scope	Project level.
Description	Specifies which underlying data type to use for the generated Visual State data type. This option requires that you have specified the option `-typestyle2`.
Example	`-typeVS_INT16int`
See also	-*typestyle*, page 746.

Project>Options>Code Generation>*project*>Types>Type to use for VS_*

-typeheaderfile

Syntax	`-typeheaderfile`*`path`*
Parameters	*`path`* The file path to the file to include.
Scope	Project level.
Description	Specifies a header file that will be included in all files that need type definitions to declare manually specified underlying data types for the generated Visual State data types. This option requires that you have specified the option `-typestyle2`.
See also	-*typestyle*, page 746.

Project>Options>Code Generation>*project*>Types>File to #include that will provide typedefs for the types specified manually

-typestyle

Syntax: `-typestyle{0|1|2}`

Parameters:

`0` (default)	Uses the standard Visual State data types.
`1`	Uses C99 data types, where possible, as the underlying types for the generated `VS_*` types.
`2`	Allows you to specify individually for each generated `VS_*` type which underlying data type to use, using one of the options `-typeheaderfile` or `-type`*`VStype`*.

Scope Project level.

Description Selects the underlying data types for the generated Visual State data types.

Project>Options>Code Generation>*project*>Types>Types style

-useapiprefix

Syntax `-useapiprefix{0|1}`

Parameters

`0`	No prefix is used in front of identifiers, functions, etc.
`1` (default)	A prefix is used in front of all identifiers, functions, etc.

Scope System level.

Description Determines whether the Classic Coder uses the prefix specified with the `-apiprefix` option in front of all identifiers, functions, etc, in the system.

Project>Options>Code Generation>*system*>API Functions>Use prefix for API

-useautovariables

Syntax `-useautovariables{0|1}`

Parameters

`0`	Auto variables are not allowed in the generated API code.
`1` (default)	Auto variables are allowed in the generated API code.

Scope System level.

Description Determines whether auto variables are allowed in the generated API code. Allowing auto variables might make the API code faster but it can also lead to increased stack usage.

Project>Options>Code Generation>*system*>Code>Use auto variables

-useguardtypecast

Syntax

`-useguardtypecast{0|1}`

Parameters

`0`	The Classic Coder does not use guard type casts.
`1` (default)	The Classic Coder uses guard type casts.

Scope

System level.

Description

Determines whether the Classic Coder uses guard type casts.

Project>Options>Code Generation>*system*>Code>Use guard type cast

-useheap

Syntax

`-useheap{0|1}`

Parameters

`0`	The Classic Coder does not use heap memory.
`1` (default)	The Classic Coder uses heap memory.

Scope

Project level.

Description

Determines whether the Classic Coder uses 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.

Project>Options>Code Generation>*project*>Code>Use heap memory

-uselivesamplingbuffer

Syntax

`-uselivesamplingbuffer{0|1}`

Parameters

`0`	Prevents C-SPYLink from reading data from the sampling buffer while the target application is executing.

`1` (default)	Enables C-SPYLink to read data from the sampling buffer while the target application is executing.

Scope

System level.

Description

Determines whether C-SPYLink can read data from the sampling buffer while the target application is executing. The target controller must support live read.

Project>Options>Code Generation>*system*>C-SPYLink>Enable sampling buffer readout

-usepop

Syntax

`-usepop{0|1}`

Parameters

`0`	The Classic Coder uses the output path specified by the `-spath` option for system files.
`1` (default)	The Classic Coder uses the same output path for system files as the path specified for all project files.

Scope

System level.

Description

Determines whether the Classic Coder uses the same output path for system files as the path specified for all project files.

Project>Options>Code Generation>*project*>File Output>Use Project output path

-userecordingbuffer

Syntax

`-userecordingbuffer{0|1}`

Parameters

`0`	Disables the recording buffer.
`1` (default)	Enables the recording buffer.

Scope

System level.

Description

Determines whether to use a recording buffer to make it possible to make recordings (execution logs) at almost full speed. Enabling the buffer also makes it possible to display sampling backups. Use the option `-recordingbuffersize` to set the size of the buffer.

See also

-recordingbuffersize, page 734.

Project>Options>Code Generation>*system*>C-SPYLink>Enable recording buffer

-userfileinclusion

Syntax

`-userfileinclusion`*`path`*

Parameters

`path`	The file path to the file to include.

Scope

Project level.

Description

Specifies a file to include in every generated source file.

Project>Options>Code Generation>*project*>File Output>File that will be included verbatim in each generated source file

-userlkw

Syntax

`-userlkw{0|1}`

Parameters

`0` (default)	Disables additional RealLink extended keywords.
`1`	Enables additional RealLink extended keywords.

Scope

Project level.

Description

Specifies whether to use additional RealLink extended keywords.

See also

Debugging design models using RealLink, page 785.

Project>Options>Code Generation>*project*>RealLink>Use additional RealLink extended keywords

-usesamplingbuffer

Syntax `-usesamplingbuffer{0|1}`

Parameters

`0` (default)	Disables on-target sampling buffers for a single macrostep.
`1`	Enables on-target sampling buffers for a single macrostep.

Scope System level.

Description Controls on-target sampling buffers for a single macro step. If you specify `-usesamplingbuffer1`, C-SPYLink can extract large amounts of debug information from your model. This causes an increase in code size and a small reduction in execution speed. If sequence recording is used, the speed reduction will be larger. Use the option `-samplingbuffersize` to set the size of the buffer.

See also -*samplingbuffersize*, page 735.

Project>Options>Code Generation>*system*>C-SPYLink>Enable sampling buffer

-V

Syntax `-V`*`system`*

Parameters

`system`	The name of a system.

Scope System level.

Description Specifies the system that the following system options apply to. System options that are specified before any system has been specified apply to all systems.

This option is not needed in the graphical interface.

-variant

Syntax `-variant`*`name`*

Parameters

`name`	The name of the variant.

Scope Project level.

Description Specifies which variant to generate code for. By default, the Coder generates code for the complete model.

See also *Using variants and features*, page 217.

Use the **Variant** toolbar.

-vsbooltype

Syntax `-vsbooltype`*`datatype`*

Parameters

`datatype`	The data type to use for the `VS_BOOL` type at runtime. By default, the value is `int`.

Scope Project level.

Description Specifies the data type to use for the `VS_BOOL` type at runtime.

Project>Options>Code Generation>*project*>Style>VS_BOOL type

-vsdeduct

Syntax `-vsdeduct{0|1}`

Parameters

`0` (default)	Disables the generation of the `VSDeduct` function.
`1`	Enables the generation of the `VSDeduct` function.

Scope System level.

Description — Enables or disables the generation of the system-specific `VSDeduct` function rather than generating the function with the name `SEM_Deduct`/`SMP_Deduct`. Setting this option to `1` makes it easier to switch between the different APIs and simplifies the code that must be user-written.

Project>Options>Code Generation>*system*>API Functions

-vselementexpl

Syntax — `-vselementexpl{0|1}`

Parameters

`0` (default)	Disables the generation of the `VSElementExpl` function.
`1`	Enables the generation of the `VSElementExpl` function.

Scope — System level.

Description — Enables or disables the generation of the system-specific `VSElementExpl` function rather than generating the function with the name `SEM_Expl`/`SEM_ExplAbs`/`SMP_Expl`/`SMP_ExplAbs`. Setting this option to `1` makes it easier to switch between the different APIs.

Project>Options>Code Generation>*system*>API Functions

-vselementname

Syntax — `-vselementname{0|1}`

Parameters

`0` (default)	Disables the generation of the `VSElementName` function.
`1`	Enables the generation of the `VSElementName` function.

Scope — System level.

Description — Enables or disables the generation of the system-specific `VSElementName` function rather than generating code that relies on some macros. Setting this option to `1` reduces the number of generated macros and makes the resulting code easier to read.

Project>Options>Code Generation>*system*>API Functions

-vsinitall

Syntax `-vsinitall{0|1}`

Parameters

`0` (default)	Disables the generation of the `VSInitAll` function.
`1`	Enables the generation of the `VSInitAll` function.

Scope System level.

Description Enables or disables the generation of the system-specific `VSInitAll` function. Setting this option to `1` makes it easier to switch between the different APIs.

Project>Options>Code Generation>*system*>API Functions

-warnings_affect_exit_code

Syntax `-warnings_affect_exit_code{0|1}`

Parameters

`0` (default)	Warnings generate a zero exit code.
`1`	Warnings generate a non-zero exit code.

Scope Project level.

Description By default, the exit code is not affected by warnings, because only errors produce a non-zero exit code. This option determines whether warnings also generate a non-zero exit code.

Project>Options>Code Generation>*project*>Configuration>Warnings affect exit code

-warnings_are_errors

Syntax `-warnings_are_errors{0|1}`

Parameters

`0` (default)	Warnings are treated like warnings.
`1`	All warnings are reclassified as errors.

Scope Project level.

Description Determines whether all warnings are reclassified as errors. If the Classic Coder encounters an error, no code is generated.

Project>Options>Code Generation>*project*>Configuration>Treat warnings as errors

-wrapperfunctionkeyword

Syntax `-wrapperfunctionkeyword`*`keyword`*

Parameters

`keyword`	A string that will be used as a keyword.

Scope System level.

Description Specifies an extended keyword to be used for all generated wrapper functions for guards and action calls.

Project>Options>Code Generation>*system*>Ext. Keywords>Extended keyword to use on generated wrapper functions

Part 7. Testing your state machine model on hardware

This part of the *IAR Visual State User Guide* includes these chapters:

- Debugging design models using C-SPYLink
- Debugging design models using RealLink

Debugging design models using C-SPYLink

- Introduction to debugging using C-SPYLink
- Debugging using C-SPYLink
- Graphical environment for C-SPYLink

Introduction to debugging using C-SPYLink

Learn more about:

- *Briefly about C-SPYLink*, page 759
- *Operating overview*, page 760
- *C-SPYLink debugging resources*, page 760
- *C-SPYLink execution modes*, page 762
- *State machine breakpoints*, page 766
- *Execution sequences*, page 768

BRIEFLY ABOUT C-SPYLINK

C-SPYLink connects IAR Visual State and IAR Embedded Workbench® to make true high-level state machine debugging possible directly in C-SPY, in addition to the normal C level symbolic debugging. This means that you can debug your state machine model on target hardware.

C-SPYLink provides these main features:

- Live monitoring of the complete global state of the state machine model
- State machine level breakpoints; breakpoints can also be set on specific events, signals, or state configurations
- A choice between running the target at full speed with small overhead and with visual feedback, if target permits, or balancing between speed and feedback if hardware limits the possibilities.
- No extra user-written support code for communication, configuration of port protocols, etc., is needed.
- Graphical animation of your debug session, see *Graphical animation*, page 335.

C-SPYLink consists of two parts:

- A plugin module for C-SPY and the IAR Embedded Workbench IDE
- Extra instrumentation code and meta data required for the debug session.

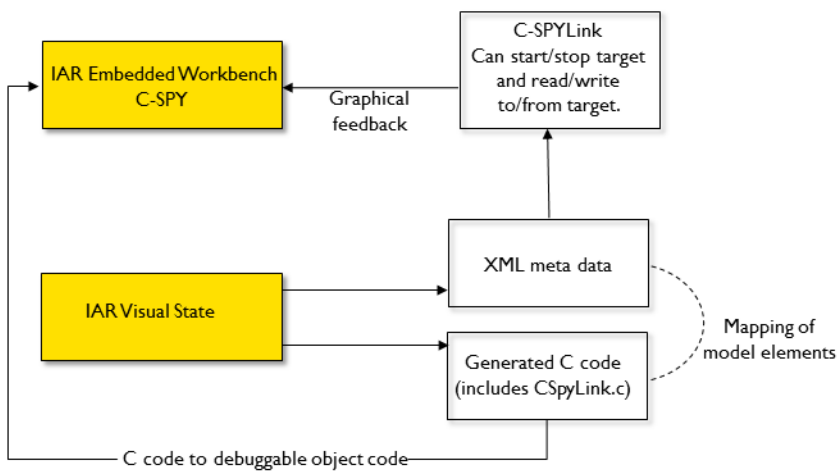
C-SPYLink requirements

To take full advantage of C-SPYLink, you need:

- A copy of IAR Embedded Workbench with an IDE of version 4.1 or later. You will find the version number by choosing **Help>About>Product Info** in the IDE.
- For hardware debugging, you need hardware debug support. For example, I-Jet or a general JTAG probe, NEXUS® or hardware emulator support and the corresponding C-SPY driver for the debug system.

OPERATING OVERVIEW

This figure shows the operating principles behind C-SPYLink:



Using C-SPYLink in your development project is very straightforward. See *Before starting the debug session*, page 770.

C-SPYLINK DEBUGGING RESOURCES

To debug your state machine model in the C-SPY debugger, the Coder must generate some extra instrumentation code to handle:

- breakpoints
- sampling buffers

- recording buffer
- full instrumentation code.

Note that including code for these resources affects both the execution speed and size of the application, both in terms of flash memory and RAM use.

To make the Coder generate code for handling the debugging resources, you must specify certain settings to the Navigator. For more information, see *Before starting the debug session*, page 770.

The breakpoint resources

C-SPYLink requires one breakpoint on the target hardware system. This breakpoint can either be a *hardware breakpoint* or a *shared DLIB breakpoint* (or if Arm is used, the latter is instead an *Arm EABI shared semihosting breakpoint*).

The breakpoint that C-SPYLink uses will be overloaded with one or more logical breakpoints—*state machine breakpoints*—that you can set during your debug session.

State machine breakpoints consist of both:

- Data structures that store information about the breakpoint.
- Instrumentation code that compares the content of the data structures with the current situation in the state machine at the break.

C-SPYLink also uses state machine breakpoints internally, which are placed at appropriate places to gather information needed for the debug session.

See also *State machine breakpoints*, page 766.

The sampling buffers

C-SPYLink uses two to three buffers on the target hardware to collect information about the macrosteps being executed.

If you use the sampling buffers, execution speed will be a little slower and the code size will increase.

The sampling buffers are allocated on a per-system basis. If your Visual State project has more than one system that will run in the same application, you can decide per system whether you want to have the sampling buffer generated. However, when you set up for an execution mode for your debug session that uses the sampling buffers, that mode applies to all systems that were generated for using a sampling buffer.

When you allocate the size for the buffers in your linker configuration file, you can start by estimating the size. The linker generates an error if the sampling buffers are too large for your available memory.

The recording buffer

If you have RAM available on your target, you can allocate some of it for a recording buffer. This buffer can be used for recording execution sequences. See also *Execution sequences*, page 768.

Full instrumentation code

The instrumentation code is code that the Coder inserts in certain positions in the Coder-generated code, and which is used for managing breaks and required information at certain situations.

Full instrumentation code is mainly intended to be used when you do not have enough space for sampling buffers.

C-SPYLINK EXECUTION MODES

C-SPYLink can operate in various *execution modes*, with different behavior and impact on real-time performance, typically execution speed and level of information status.

Which execution mode you decide to use depends on the available debugging resources you have, see *C-SPYLink debugging resources*, page 760. If you select an execution mode for which you do not have the required resources, C-SPYLink will issue a warning.

In short, these execution modes are available:

- Full speed—full information continuously updated
- Full speed—full information at stops
- Medium speed—information at stops and based on snapshots
- Low speed—full information continuously updated

For information about when and how to specify the execution mode, see *Before starting the debug session*, page 770.

Full speed—full information continuously updated

In this mode, your application will execute at full speed without any stops initiated by C-SPYLink. C-SPY windows are updated continuously.

Using this mode, only on-target breakpoints can be set, see *Types of state machine breakpoints*, page 766.

To use this mode, use these option settings and menu commands:

Options/commands in	Setting
Coder Options dialog box	**Enable sampling buffer live readout**
	Sample buffer size: specify the required buffer size
	Number of state machine breakpoints: specify the required number of hardware breakpoints
Visual State menu in the IAR Embedded Workbench IDE	**Instrumentation>Full speed, sampling buffer capture**
	Sampling Buffer Capture Setting>Live

Table 34: Setting up for execution mode, alternative 1

Full speed—full information at stops

In this mode, your application will execute at full speed without any stops initiated by C-SPYLink. No feedback is provided until execution stops, by means of a breakpoint or Ctrl+C. When this happens, C-SPYLink will update the affected windows with the current system state.

Using this mode, only on-target breakpoints can be set, see *Types of state machine breakpoints*, page 766.

To use this mode, use these option settings and menu commands:

Options/commands in	Setting
Coder Options dialog box	**Enable sampling buffer**
	Sample buffer size: specify the required buffer size
	Number of state machine breakpoints: specify the required number of hardware breakpoints
Visual State menu in the IAR Embedded Workbench IDE	**Instrumentation>Full speed, sampling buffer capture**

Table 35: Setting up for execution mode, alternative 2

Medium speed—information at stops and based on snapshots

This mode provides reduced execution speed. C-SPYLink uses the sampling buffer to collect the information at each macrostep without stopping. Execution is periodically stopped in the background to read out information and update the displayed information. When viewing the state machine model in C-SPYLink, the hardware seems to be executing. In reality, the hardware has temporarily stopped at regular intervals.

If it is critical for your hardware that the execution must not stop, do not enable this mode.

To use this mode, use these option settings and menu commands:

Options/commands in	Setting
Coder Options dialog box	**Enable sampling buffer**
	Sample buffer size: specify the required buffer size
	Number of state machine breakpoints: specify the required number of hardware breakpoints
Visual State menu in the IAR Embedded Workbench IDE	**Instrumentation>Full speed, sampling buffer capture**
	Sampling Buffer Capture Setting>Periodic Stop

Table 36: Setting up for execution mode, alternative 3

Full speed—no feedback, alternated with information at stops

Initially, this mode provides full execution speed but without information feedback. However, you can choose to stop at a certain location, for example by setting a state machine breakpoint. When stopped, you can change to **Slow speed, Full instrumentation** and you will get detailed information at slow speed.

To use this mode, use these option settings and menu commands:

Options/commands in	Setting
Coder Options dialog box	**Number of state machine breakpoints:** specify the required number of hardware breakpoints
Visual State menu in the IAR Embedded Workbench IDE	**Instrumentation>Full speed, No instrumentation** alternated with **Instrumentation>Low speed, Full instrumentation**

Table 37: Setting up for execution mode, alternative 4

Low speed—full information continuously updated

In this mode, your application will run at low speed but you will get a very fine-grained control over what is happening on the target controller at any given point in time. The C-SPY windows are continuously updated with detailed information. For each event, you can see which action functions are called and their argument list.

In this mode, the synchronization hardware breakpoint is overloaded with several internal state machine breakpoints. Each time such a breakpoint is triggered, data about the system state is read. The continuous stopping and restarting of execution has a severe negative impact on runtime performance, which might be a problem.

The only extra cost in terms of memory, both ROM and RAM, for this mode is the calls to the breakpoint function, which are few. Further, this mode requires some instrumentation code overhead and negligible RAM overhead. The actual overhead depends on the target CPU.

To use this mode, use these option settings and menu commands:

Options/commands in	Setting
Coder Options dialog box	**Enable full instrumentation**
Visual State menu in the IAR Embedded Workbench IDE	**Instrumentation>Low speed, Full instrumentation**

Table 38: Setting up for execution mode, alternative 5

Hints for choosing the most useful execution mode

These are some guidelines for setting up an efficient execution mode that suits your needs based on your available resources:

- If your target hardware supports breakpoints and sampling buffers, you would like to use them because that gives you an efficient balance between:
  - high execution performance
  - good information feedback
  - the possibility to set state machine breakpoints on target
- If you have a limited set of breakpoints and limited space for the sampling buffer, you can still use both of these even though they are limited. In this case, you can set fewer state machine breakpoints on target and you must reduce resolution for the information feedback.
- If you do not have support for breakpoints and memory space for the sampling buffer on your target, you must make a choice because you cannot get both high execution performance and full information. You can achieve:
  - Full speed, but without information.

    However, if you can set breakpoints, you can stop execution but without getting feedback about the situation in the state machine (except for the triggered breakpoint).
  - Very slow execution but with full information.

  Note that you can alternate between these two alternatives.
- If you want to record an execution sequence, you can use a dedicated recording buffer at full speed, the sampling buffer at reduced speed, or full instrumentation at very low speed. See *Execution sequences*, page 768.

STATE MACHINE BREAKPOINTS

Using *state machine breakpoints*, you can specify a set of goal states from different parallel regions of your state machine model. Execution will then stop when the breakpoint states are all active at the same time. You can also specify an event or a signal as a breakpoint condition.

When a breakpoint is triggered, there are three visual clues to highlight the breakpoint:

- The breakpoint number in the **Breakpoints** window is blinking.
- A message in the **Debug Log** window says that a state machine breakpoint has been triggered.
- The edit window displays a green arrow on the `_VS_breakpoint` function. This function is used by C-SPYLink as a placeholder for the real C-SPY breakpoint used by IAR Visual State to synchronize data. This visual clue is not displayed if the breakpoint is a shared DLIB breakpoint or an Arm EABI semi-hosting breakpoint, see *Using shared DLIB breakpoints*, page 773.

Types of state machine breakpoints

There are two types of state machine breakpoints—*full instrumentation* breakpoints and *on-target* breakpoints. They have the same features, but different performances. A breakpoint can hold information about a trigger (event or signal) and state vector before and after a step.

Full instrumentation breakpoints:	For **Low speed, Full instrumentation**, all breakpoints will be treated as full instrumentation breakpoints. They do not take up any extra memory, because C-SPYLink handles all checking of breakpoint conditions. There is no limit to the number of full instrumentation breakpoints.

On-target breakpoints:	When anything else than **Low Speed, Full instrumentation** is enabled, all breakpoints will be regarded as on-target breakpoints if you have allocated space for the breakpoint buffer. A *breakpoint buffer* is created in target memory and a small amount of code is generated to check the breakpoint conditions. In the Navigator, to allocate the necessary space in target memory, use the **Number of state machine breakpoints** option in the **Coder Options** dialog box (on the **C-SPYLink** page). In C-SPY, these breakpoints can be used with or without the sampling buffer. Without the sampling buffer, the C-SPY windows will not be updated when execution stops. On-target breakpoints can have a status message next to them in the **Breakpoints** window.

Pre- and post-deduct conditions

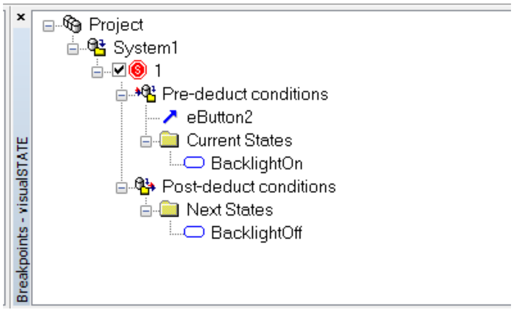
A breakpoint can be set to trigger at two different occasions: before and after an event or signal has been processed.

Pre-deduct condition:	Stops execution *before* processing (deduction of) a new trigger, but after the complete processing of the preceding microstep. This means that it is the result of the previous microstep processing that will be used as the stop criteria.
Post-deduct condition:	Stops execution *after* the event processing (deduction) is complete.

A minor difference between these is that a pre-deduct condition is tested when the trigger is injected. The real difference is seen when the pre-deduct condition is used in combination with other conditions, such as a trigger or a second state condition at the post-deduct node.

If you want the breakpoint to trigger when the execution passes from one specified state configuration to another specified configuration, you can add the precondition states as a pre-deduct condition and the postcondition states as a post-deduct condition.

The breakpoint in this example is triggered when the `BackLightOn` state is active. The event `ev_BUTTON2` is processed and the resulting state is `BackLightOff`:



For more information, see:

- *Using state machine breakpoints*, page 772
- *Using shared DLIB breakpoints*, page 773
- *Breakpoints window*, page 780.

EXECUTION SEQUENCES

To help you debug state machines, you can record an execution sequence of signals, events, actions, changes to variables (requires full instrumentation code), etc, and save the sequence to an XML file. This XML file can be loaded in the Validator. A maximum of 100,000 steps can be recorded.

Sequences are recorded with one of these methods:

Recording buffer: The execution runs at almost full speed on the target hardware. The target hardware must have enough RAM to record the sequence.

In the Navigator, allocate a buffer by choosing **Project>Options>Code Generation>C-SPYLink>Recording buffer size** and specifying a buffer size.

Sampling buffer: The recording is performed by stopping the execution after each macrostep to read out the sampling buffer. This slows down execution considerably more than using the recording buffer, but it requires no extra on-target memory except for the sampling buffer.

This method is faster than using Full instrumentation.

Full instrumentation: The execution stops frequently. This allows reading out sequences with no extra on-target memory required, but execution is much slower.

Note that this method can handle internal variables.

To enable recording execution sequences, see *Recording an execution sequence*, page 774.

Debugging using C-SPYLink

What do you want to do?

See also *Animating debug sessions graphically*, page 335.

INSTALLING C-SPYLINK

Support for debugging your state machine model using C-SPY is automatically provided when you install IAR Visual State by means of `ValidatorCSpy.dll`.

This DLL file can interact with the debugger to read and write data on the target controller or in the C-SPY Simulator. The file can also control the execution of the application on the target hardware or in the simulator.

To install additional C-SPYLink files:

1. In your IAR Embedded Workbench IDE, choose **Help>About>Product Info**. Note which version number that is listed for **IAR Embedded Workbench common components**, and remember it.
2. In the `Visual State\plugin` directory (in your IAR Visual State product installation), click the `EWx` directory that matches the version number of the common components of your IAR Embedded Workbench.
3. In the `EWx` directory, you will find the C-SPYLink plugin module `vs.ewplugin`, an XML file that points to the `ValidatorCSpy.dll` file in the Visual State installation directory. Copy the `vs.ewplugin` file to the `common\plugins` directory of your IAR Embedded Workbench product installation.

When IAR Visual State is installed, it searches for all IAR Embedded Workbench products that can support C-SPYLink and installs the plugin module in the `common\plugins` directory for each product version. In addition, a copy of the `vs.ewplugin` file will be placed in the `Plugin` directory of the IAR Visual State product installation.

4 If you install another IAR Embedded Workbench product version after you have installed IAR Visual State, you must copy this `vs.ewplugin` file to the IAR Embedded Workbench `common\plugins` directory of the new product. If you run into problems when you install several versions:

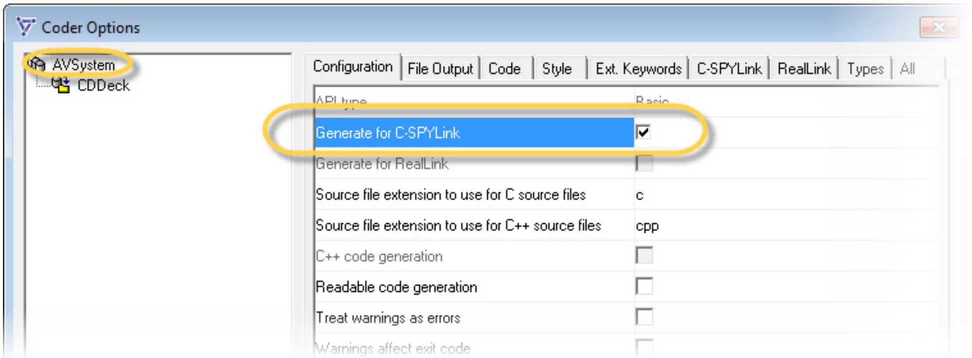
- Make sure that the file path between the `<dllFile>` and `</dllFile>` tags in the `vs.ewplugin` file matches your installation location for IAR Visual State.
- Make sure that the name of the `ValidatorCSpy` file in the `vs.ewplugin` file reflects your Embedded Workbench version.

BEFORE STARTING THE DEBUG SESSION

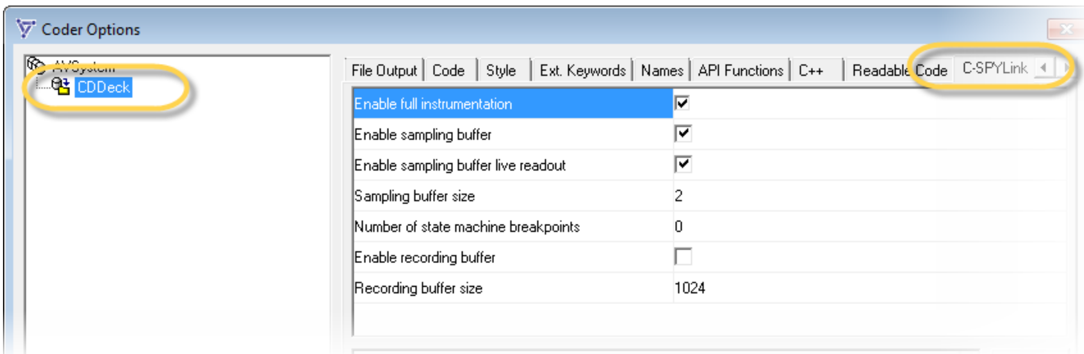
Before you can debug your design model in C-SPY, you must enable C-SPYLink in both the IAR Embedded Workbench IDE and in IAR Visual State.

1 In the Navigator, choose **Project>Options>Code generation** to open the **Coder Options** dialog box.

2 In the left-hand pane, select the project and then select the **Generate for C-SPYLink** option.



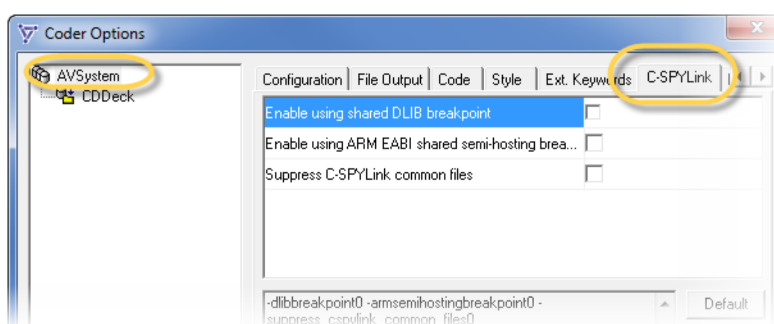
3 In the left-hand pane, select the system you want to debug and click the **C-SPYLink** tab.



For information about how to set up an efficient execution mode, see *C-SPYLink execution modes*, page 762.

For reference information about the options, see *Classic Coder Options dialog box : C-SPYLink*, page 692.

4 In the left-hand pane, select the project you want to debug and click the **C-SPYLink** tab.



For information about breakpoints, see *State machine breakpoints*, page 766.

For reference information about the options, see *Classic Coder Options dialog box : C-SPYLink*, page 692.

Click **OK** when you are finished.

5 In the IAR Embedded Workbench IDE, choose **Project>Options>Debugger>Plugins** and enable the C-SPYLink plugin module. Start your debug session.

The **Visual State** menu is now available in the IAR Embedded Workbench IDE.

6 Choose **Visual State>Instrumentation** and choose the alternatives that suits your requirements. For guidelines, see *C-SPYLink debugging resources*, page 760 and *C-SPYLink execution modes*, page 762.

USING STATE MACHINE BREAKPOINTS

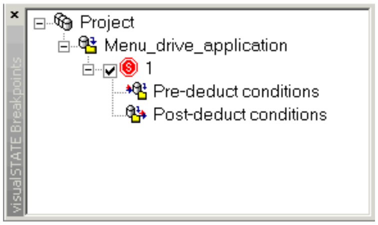
1 In the Navigator, choose **Project>Options>Code generation>C-SPYLink** and select either the option **Enable full instrumentation** or specify **Number of state machine breakpoints** to be more than 0.

2 In the IAR Embedded Workbench IDE, choose **Visual State>Instrumentation>Low Speed, Full Instrumentation** if you have specified breakpoints to be 0. If you have specified **Number of state machine breakpoints** to be more than 0, you can choose any of the options **Full speed, No Instrumentation**, or **Full speed, Sampling Buffer Capture**.

3 Choose **Visual State>View>Breakpoints** to open the **Breakpoints** window and make sure you have the windows open that display the types of breakpoints triggers you want to use. In this example, the **States** window is used.

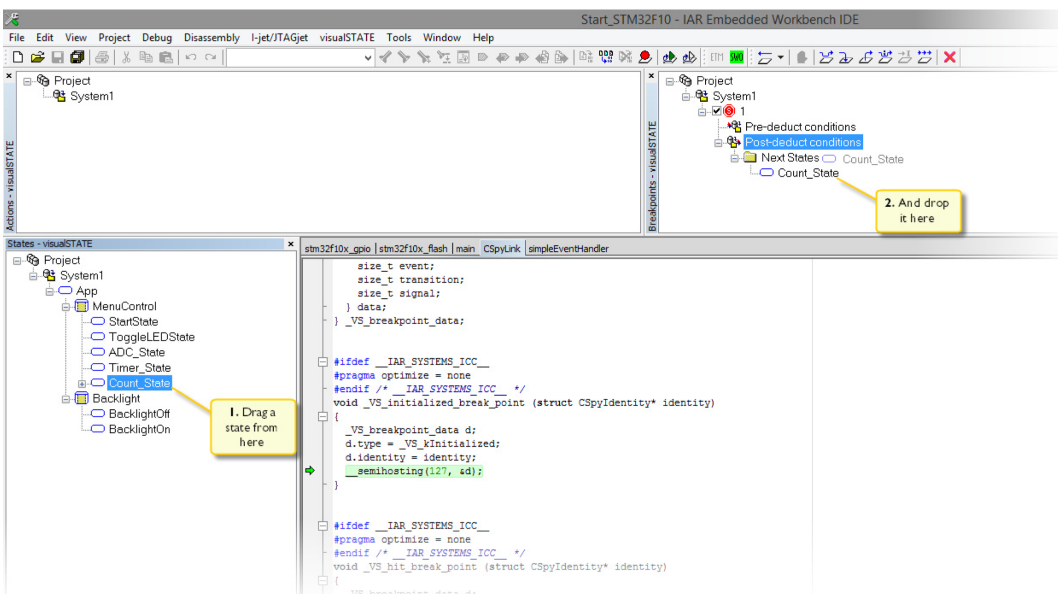
4 To create a new breakpoint, right-click the system name node in the **Breakpoints** window and choose **New Breakpoint**.

The new breakpoint will look like this:



To enable a breakpoint, use the checkbox to the left of the breakpoint. You can make it trigger at two different occasions: before and after an event or signal has been processed. See *Pre- and post-deduct conditions*, page 767.

5 Add conditional triggers to the breakpoint by dragging elements from other windows. For example, create a post-deduct state condition by dragging one or more states from the **States** window to the post-deduct node of the breakpoint.



6 Choose **Debug>Go** to start the execution and watch what happens when the breakpoint is triggered.

When you have examined the state of the system, you can continue execution as usual.

USING SHARED DLIB BREAKPOINTS

Normally, C-SPYLink allocates a breakpoint that is shared by all C-SPYLink debugging features. If you are using the IAR DLIB runtime environment, you can instead use a shared DLIB breakpoint to make C-SPYLink share the same breakpoint as the C library code for debugging.

To use a shared DLIB breakpoint:

1 In the Navigator, choose **Project>Options>Code Generation**, select the project in the left-hand pane and click the **C-SPYLink** tab.

2 Select **Enable using shared DLIB breakpoint**.

For IAR Embedded Workbench for Arm 5.1 and later, there is another shared breakpoint—**Enable using ARM EABI shared semi-hosting breakpoint**—that can be enabled in a similar manner.

This allows you to save a breakpoint by overloading a state machine breakpoint on a shared debug breakpoint.

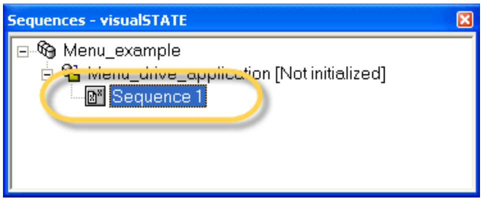
RECORDING AN EXECUTION SEQUENCE

1 There are different mechanisms for recording an execution sequence. Before you can record you must set up for it, and how you do that depends on which debugging resources you have. Choose between:

- If possible, use the recording buffer. In the IAR Embedded Workbench IDE, choose **Visual State>Sequence>Recording Buffer**.
- If a recording buffer is not available, the mechanism will automatically depend on the execution mode you are using. Note that if you are using the sampling buffers, the execution speed will decrease while you are recording the execution sequence.

2 Choose **Visual State>View>Sequence** to open the **Sequences** window.

3 In the **Sequences** window, select the appropriate system, right-click and choose **New Sequence** from the context menu. A **Sequence1** label appears in the window.



4 Select **Sequence1**, right-click and choose **Start Recording** from the context menu.

5 Debug your state machine.

6 When finished, right-click **Sequence1** in the **Sequences** window and chose **End Recording** from the context menu.

7 If you want, save your sequence to a file. Right-click and choose **Save** from the context menu.

TROUBLESHOOTING—USING C-SPYLINK

This is a list of issues that might arise when you use C-SPYLink:

- If code is running from flash memory and the hardware or the low-level debug driver does not support code breakpoints in flash memory, Full instrumentation mode and other breakpoint-dependent features will not work. Instead, build your application for execution in RAM.
- If the available breakpoints are already used by other C-SPY functionality, C-SPYLink will not function properly.

  Here are some examples of breakpoint use that are not obvious:

  - I/O emulation in C-SPY needs one breakpoint to function properly. If you are using the DLIB runtime environment, you can make an extra breakpoint available by enabling the shared DLIB breakpoint or Arm EABI semi-hosting breakpoint.
  - If there is no breakpoint available, a workaround is to turn off I/O emulation on the **Linker** option page and link your own low-level implementation of the functions `putchar` and `getchar` if there are calls to any standard C library I/O in your application.
  - The **Run to main** option on the debugger options **Setup** page requires a breakpoint. Deselect this option.
  - Some other C-SPY plugin modules might also need to set a breakpoint. Disable all other plugin modules and try again.

For more information about breakpoint consumers, see the *C-SPY Debugging Guide* provided with IAR Embedded Workbench. See also *Using shared DLIB breakpoints*, page 773.

Graphical environment for C-SPYLink

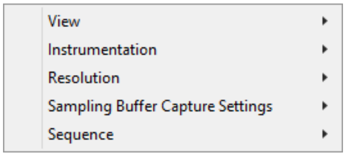
Reference information about:

These windows are available from the **Visual State** menu in the IAR Embedded Workbench IDE, when the IAR Embedded Workbench IDE is connected to a Visual State project via the C-SPYLink plugin.

See also *Designer windows in Graphical Animation mode*, page 337.

Visual State menu

The **Visual State** menu—in the IAR Embedded Workbench IDE—provides commands for using C-SPYLink to debug your state machine model in C-SPY:



Menu commands

These commands are available on the menu:

View

Displays a submenu from where you can open the windows specific to C-SPYLink. See:

Actions window, page 779

Breakpoints window, page 780

Designer windows in Graphical Animation mode, page 337

Sequences window, page 781

Signal Queues window, page 387

States window, page 782

Triggers window, page 783

Instrumentation

Displays a submenu where you can choose between:

Full speed, No Instrumentation	Your application will run at full speed, without any stops initiated by C-SPYLink. Only on-target breakpoints can be set and recording an execution sequence can only be performed using the **recording buffer**.
Full speed, Sampling Buffer Capture	The sampling buffer is used. This option requires that you have selected **Enable sampling buffer** in the **Coder Options** dialog box.
Low speed, Full Instrumentation	Your application will run at very low speed but you will get a very fine-grained control over what is happening on the target controller at any given point in time. The C-SPY windows are continuously updated with detailed information. For each event, you can see which action functions are called and their argument list. This option requires that you have selected **Enable full instrumentation** in the **Coder Options** dialog box.

All instrumentation levels will affect execution speed compared to not using any instrumentation code at all.

See also *C-SPYLink debugging resources*, page 760 and *C-SPYLink execution modes*, page 762.

Resolution

Displays a submenu where you can choose for which elements you want information available during your debug session. The more elements you choose, the more memory space is required for your buffers. Choose between: Actions, Fired Signals, States, Transitions, Variables.

Note that Variables can only be used if you have selected **Enable Full Instrumentation** in the **Coder Options** dialog box.

Sampling buffer Capture Settings

Displays a submenu where you can choose between:

Live	C-SPYLink reads the sampling buffer without stopping the target. Whether this is possible or not depends on the debug probe you are using. This capture mode requires using the **Enabling sampling buffer readout** option. If this mode is selected but not supported by the probe, C-SPYLink issues a warning, and the feature is disabled.
Periodic Stop	C-SPYLink stops at pre-configured intervals. At each stop, the current completed part of the sampling buffer is read.
Delay	Choose a delay in seconds.

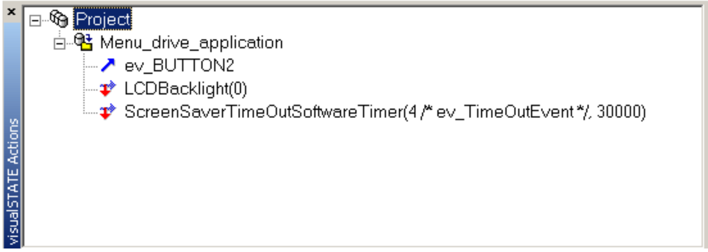
Sequence

Displays a submenu where you can choose between:

Recording buffer	Uses the recording buffer while recording an execution sequence, see *The recording buffer*, page 762.
Record All Systems	Records all systems.
End All Recording	Ends all recording.
Delete All	Deletes all recorded information.

Actions window

The **Actions** window is available from the **Visual State>View** submenu in the IAR Embedded Workbench IDE.



This window contains information about a step.

Display area

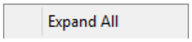
The display area shows:

- the action functions that are executed as a result of event processing (with parameters but not with variable arguments) and the event or signal that caused the processing
- transitions
- assignments (internally generated action functions)

When you single step through the Visual State event processing loop using the **Enable Full Instrumentation** Coder option and the **Low Speed, Full Instrumentation Visual State** menu command, the window is updated for each completed microstep.

Context menu

This context menu is available:



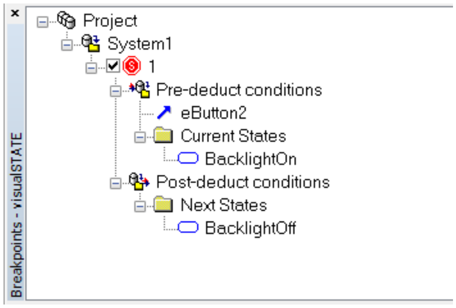
This command is available:

Expand All

Displays the complete hierarchy.

Breakpoints window

The **Breakpoints** window is available from the **Visual State>View** submenu in the IAR Embedded Workbench IDE.



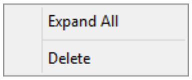
Use this dialog box to configure state machine breakpoints.

A breakpoint can be enabled and disabled with the checkbox. When the debug session is closed, the breakpoint configuration will be remembered until the next session.

See *Using state machine breakpoints*, page 772.

Context menu

This context menu is available:



Note: Depending on what you have selected in the window, some or all of these commands are available.

These commands are available:

Expand All

Displays the complete hierarchy.

New Breakpoint

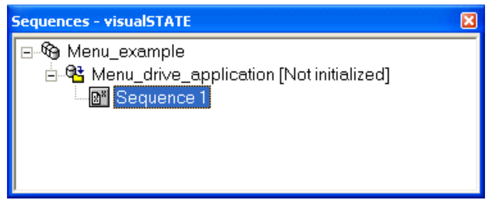
Creates a state machine breakpoint. States, events, and signals can be dragged from open windows as conditional triggers to the pre-deduct and post-deduct nodes for the breakpoint.

Delete

Deletes the selected breakpoint.

Sequences window

The **Sequences** window is available from the **Visual State>View** submenu in the IAR Embedded Workbench IDE.

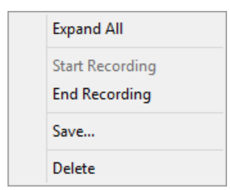


This window shows the execution sequences set up for recording, see *Execution sequences*, page 768.

If you record using the recording buffer, the window is not updated until the buffer in target memory is full or until you stop the recording. If you use the sampling buffer or full instrumentation, the window is continuously updated.

Context menu

This context menu is available:



Note: Depending on what you have selected in the window, some or all of these commands are available.

These commands are available:

Expand All

Expands a node consisting of three periods (...) to show all nodes. Nodes corresponding to up to 1,000 underlying steps are displayed with the ... node in the middle—500 steps on each side.

Start Recording

Starts the recording.

End Recording

Stops the recording.

New Sequence

Creates a new sequence.

Save

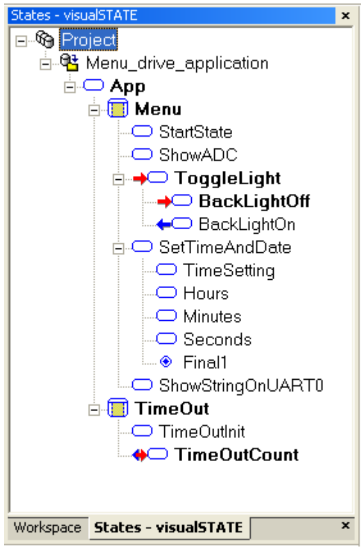
Saves the recorded sequence.

Delete

Deletes the recorded sequence.

States window

The **States** window is available from the **Visual State>View** submenu in the IAR Embedded Workbench IDE.



This window shows the complete system state configuration.

Red arrows indicate states that have become active since the last window update. For Full instrumentation this means the resulting states of the last complete event processing step.

Blue arrows indicate states that were left as the result of the last complete event processing (macrostep).

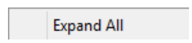
A blue arrow leaves a state and a red arrow that enters the same state indicates either that:

- The state has an internal transition or self-transition that triggered in the event processing, or that
- The state is already active and was not deactivated by the last event processing

The **States** window is a simplified representation of your state machine model. To see the model as it looks in the Visual State Designer, choose **Visual State>View>Graphical Animation**.

Context menu

This context menu is available:



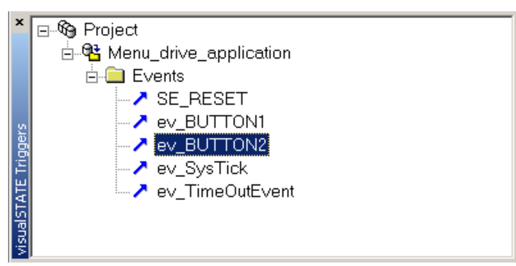
This command is available:

Expand All

Displays the complete hierarchy.

Triggers window

The **Triggers** window is available from the **Visual State>View** submenu in the IAR Embedded Workbench IDE.



This window shows all events and signal triggers for the systems that have C-SPYLink enabled. Events and signal triggers can be dragged and dropped as event conditions on breakpoints.

Context menu

This context menu is available:

This command is available:

Expand All

Displays the complete hierarchy.

Debugging design models using RealLink

- Introduction to debugging using RealLink
- Debugging using RealLink
- RealLink memory consumption
- Graphical environment for RealLink

Introduction to debugging using RealLink

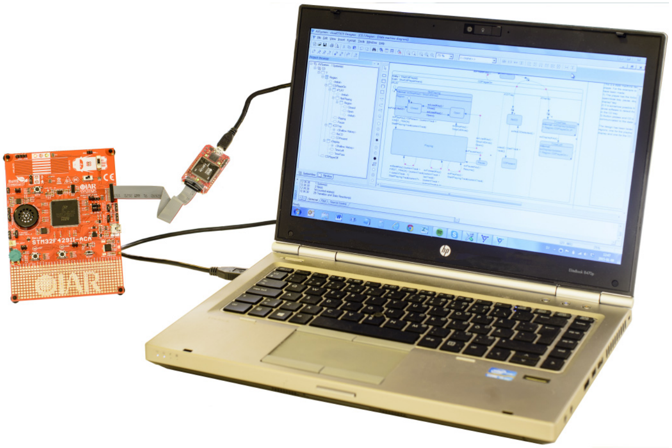
Learn more about:

- *Briefly about RealLink*, page 785
- *Visual State elements supported by RealLink*, page 787
- *Validator windows in target versus Validator mode*, page 788
- *Recorded sequences of target tests*, page 789
- *Target requirements*, page 789

BRIEFLY ABOUT REALLINK

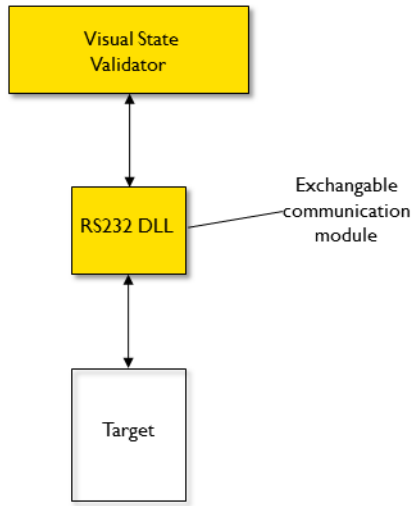
With RealLink you can monitor and control the runtime behavior of your state machine model in the target application. Typically, you can use RealLink if you have another development tool than IAR Embedded Workbench, in which case you can use C-SPYLink instead.

RealLink consists of some specific software running on the host computer, some code running on the target (some generated by the Coder and some for the communication which you should write), and a communication link between the host computer and the target.



The communication between the Validator and target is established by means of a communication module. RealLink supports multiple communication modules that each provides an interface to a specific link to the target, such as a serial connection (RS232), or a TCP/IP connection.

Each communication module automatically integrates itself with the Validator via a communication plugin module (DLL):



IAR Visual State includes these communication plugin modules for RealLink:

- RealLink RS232 communication plugin module
- RealLink TCP/IP communication plugin module.

Once the RealLink connection is established, you have full control of the Visual State model running on the target. From the Validator, events can be sent to the target, test sequence files can be recorded and played, and variables can be changed, all on actual hardware.

VISUAL STATE ELEMENTS SUPPORTED BY REALLINK

These Visual State elements can be monitored via the Validator windows:

Events	In the **Event** window, you can 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 on target. The values of guard expressions are not considered, and if the target application does not include the `SEM_Inquiry`/`SEM_GetInput` functions, all events will be marked as 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. ***TIP:*** If only a single element of an array is of interest, select this element in the **Variable** window and press Shift + F9 to display the element in the **Watch** window.
System state	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. Graphical animation (**Debug>Graphical Animation**) is also available when using RealLink. By using this option you can monitor the current states in the state machine diagrams in the Designer. See *Graphical animation*, page 335. ***TIP:*** If only a single branch of a Visual State 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 context menu to add the branch to the **System** window as a separate branch.
Signal queue	The **Signal Queue** window shows the signal queue of all Visual State systems.
Executed actions	The **Action** window lists the actions executed during the last step. This includes both executed action functions and assignments.

With RealLink you can monitor and control the behavior of all logical Visual State elements, except for these:

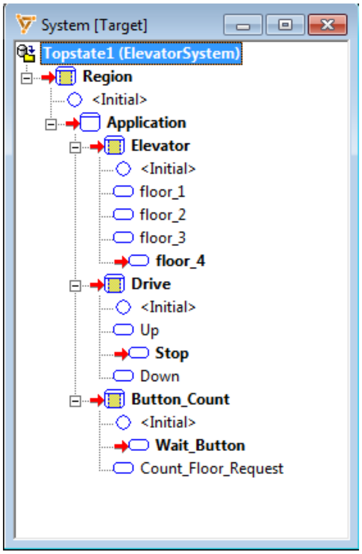
- Parameters to action functions: Their values are shown as "…" in the Validator **Action** window.
- Guard expressions of active events: The Validator **Event** window shows the active events but no guard expressions are considered. Therefore, the Validator might show an event as being active when in fact a guard expression is not satisfied.
- Instances: It is not possible to change instances from the Validator.

VALIDATOR WINDOWS IN TARGET VERSUS VALIDATOR MODE

By default, all open windows in the Validator show the Validator representation of the state machine model—the *Validator mode*. However, when the Validator is connected to target by means of RealLink, you can make the windows show the status of the model as it is on target—*Target mode*. Generally, the windows in Target mode correspond to the windows in Validator mode.

The only window that cannot be changed to showing values on target is the **Guard Expression** window.

The title bar of a window indicates which mode the design model is displayed in:



The Validator keeps track of which windows are set to target mode, and will automatically open them the next time RealLink is connected.

See also *Changing between Validator mode and Target mode*, page 799.

RECORDED SEQUENCES OF TARGET TESTS

The Validator provides commands for recording and playing test sequences. These commands are also available when running RealLink. This means that you can record a sequence executed on target and play a previously recorded sequence by means of a test sequence file. A sequence recorded on target can also be used as input to a dynamic analysis to see the test coverage.

For more information, see *Recording and playing test/event sequences*, page 349.

TARGET REQUIREMENTS

Target processors to be used with RealLink must comply with the following requirements.

Variable sizes

Variable sizes must be a multiple of 8 bits, however, max 32 bits.

Memory

Memory used by RealLink must be accessible through byte pointers. Some memory areas in specific microprocessors have only 16-bit access. These areas cannot be accessed by IAR Visual State.

RealLink requires additional memory in `CODE`, `CONST DATA`, and `DATA`. See *RealLink memory consumption*, page 802.

Communication

As part of the setup for RealLink you should write a *receive function*. 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 host computer. See *Setting up RealLink*, page 791.

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.

Visual State Uniform API requirements

If more than one Visual State 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.
- A call to `*System*VSDeduct()` must be completed before the function is called a second time.
- All systems should be running in the same task.

Debugging using RealLink

What do you want to do?

- *Controlling application execution on target*, page 801
- *Troubleshooting*, page 801

SETTING UP REALLINK

To get RealLink configured and ready for your project, these steps must be completed:

Step 1: To enable RealLink support

Step 2: To add RealLink files to your project

Step 3: To use the RealLink API

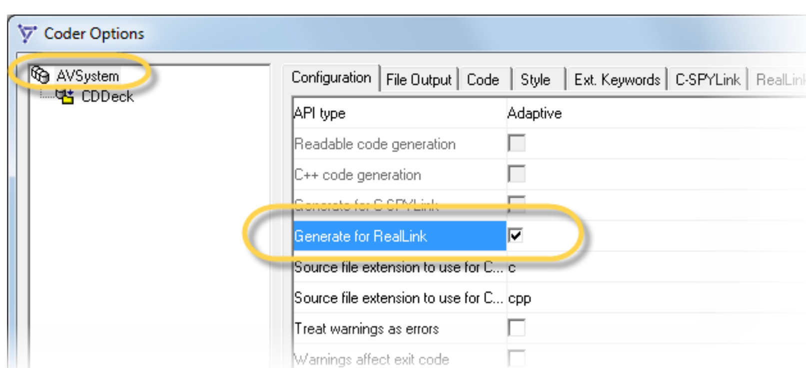
Step 4: To implement target-specific functions

Step 5: To complete the target source code

Step 6: To configure the Validator for RealLink

Step 1: To enable RealLink support:

1. In the Navigator, choose **Project>Options>Code generation** to open the **Classic Coder Options** dialog box.
2. In the left-hand pane, select the project. On the **Configuration** page, select **Generate for RealLink**.



3. On the **RealLink** page, set the options appropriate for your project.

   For reference information about the options, see *Classic Coder Options dialog box : RealLink*, page 694.

If you are using 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:

- In the **Classic Coder Options** dialog box, select **Generate for RealLink** on the **Configuration** page.
- On the **RealLink** tab, select **Use additional RealLink extended keywords**.
- Click **RealLink data extended** keywords and specify a keyword for a memory area where both read and write operations can be performed.
- Click **RealLink const data extended keyword** and specify a keyword for a memory area where read operations can be performed.

Note: When you use RealLink extended keywords, the keywords must match the Visual State Coder extended keywords. For example, the **RealLink data extended keyword** must match the keywords you specify for external and internal variables in the **Classic Coder Options** dialog box.

4 Click **OK** when you are finished.

5 On the Navigator menu, choose **Project>Code generate** to generate the source code for the active Visual State project.

Step 2: To add RealLink files to your project

1 To compile and link your project with RealLink support, you must add these two C modules to your compiler project (or makefile):

- `*System*RealLink.c`

  This file includes the C header file `*System*RealLink.h`. Make sure to include `*System*RealLink.h` in the file that contains the Visual State deduction call (a call to the Visual State API function `VSDeduct`).

  Both the `c` and the `h` files are the RealLink API files. The files are always generated by the Coder when RealLink is enabled.

- `*System*VSrlps.c`

  This file is a Coder-generated RealLink support file. You can find it in the output directory that you have specified, together with the other Coder-generated files.

The *`System`* prefix is prepended to the filename if the option **Use prefix for API** is used.

Note: Do not manually edit any RealLink files, because they will be overwritten during the next code generation.

2 For information about how to add the source files to your development project, see your compiler documentation.

Step 3: To use the RealLink API:

1 Call the Adaptive API function `SEM_InitAll`.

This replaces calls to the Adaptive 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(SEMSystem)` in the main loop but before the Visual State deduction sequence. The main loop is identified by an infinite loop, typically a `while(1)` or `for(;;)` loop.

When IAR Visual State enters the `VS_WAIT` macro, data is exchanged between the Validator and the target. When data exchange is completed, IAR Visual State resumes execution according to your commands from the Validator.

4 Below is an example of a simple Adaptive API `main` function and a simple Uniform API `main` function, both set up for RealLink. Note that the `VS_WAIT` macro is inside the main loop, but outside the deduction sequence.

Example of a `main` function using the Adaptive API and RealLink:

```
#include "SystemSEMLibB.h"
/* include RealLink API */
#include "SystemRealLink.h"

int main (void)
{
  /* Initialize the Visual State system. */
  SystemSEM_InitAll();
  /* Initializing RealLink API. */
  VS_RealLinkInit();

  while (1)/* main loop for RealLink */
  {
    unsigned char cc;
    SEM_ACTION_EXPRESSION_TYPE actionExpressNo;
    SEM_EVENT_TYPE eventNo;

    VS_WAIT(SEMSystem);
```

```
    /* deduction sequence - if we get an event */
    eventNo = GetEventFromQueue();
    if (eventNo != EVENT_UNDEFINED)
    {
      cc = SystemVSDeduct(eventNo);
      if ((cc != SES_OKAY) && (cc != SES_FOUND))
        handleError(cc);
    }
  }
  return 0;
}
```

Example of a `main` function using Uniform API and RealLink:

```
#include "RealLink.h"
/* context pointer for the system */
SEM_CONTEXT *pSEMContext = 0;

/* RL task for the system */
VS_RLTASK task;

/* Initialize this RL-task - must be before the next call */
VS_RealLinkInit(&task);

/* initialize the system */
SystemSMP_InitAll(&pSEMContext, &task);

/* main loop for RealLink with Uniform API */
while (1)
{
  unsigned char cc;
  SEM_ACTION_EXPRESSION_TYPE actionExpressNo;
  SEM_EVENT_TYPE eventNo;

  VS_WAIT(pSEMContext);

  /* deduction sequence - if we get an event */
  eventNo = GetEventFromQueue();
  if (eventNo != EVENT_UNDEFINED)
  {
    cc = SystemVSDeduct(pSEMContext, eventNo)
    if ((cc !=SES_OKAY) && (cc != SES_FOUND))
      handleError(cc);
  }
}

/* when done with the system, call this to free memory */
SMP_Free(pContext);
```

Step 4: To implement target-specific functions

Because both the Visual State API and the RealLink APIs are target-independent, they contain no information on how to use the communication device of the target.

To access the communication device, you must implement the following target-specific RealLink functions that are used by the Visual State API:

`void RealLinkReset(void)`	Resets the target. The function will be called by the RealLink API. This function might not need to do anything for your target, but you must still provide an (empty) implementation.
`void RealLinkTransmit(VS_UINT8 ch)`	Transmits one byte on the communication port or adds bytes to the buffer. The function will be called by the RealLink API.
`void TransmitFlush(void)`	This function must only be implemented if a buffer is used. The function should empty the transmit buffer.
`void Receive(void)`	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 `VS_RealLinkReceive()`.

You can change the default names of the functions by defining these macros:

```
#define VS_RL_RESET                MyReset
#define VS_RL_TRANSMIT             MyTransmit
#define VS_RL_TRANSMIT_FLUSH       MyTransmitFlush
```

Note: All Visual State systems must be located in the same task if you want to apply RealLink.

2 To implement your functions, use this as an example for how to implement a 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.

The functions in the example are for the ARM7 – LPC2138 microprocessor and the IAR Embedded Workbench for Arm compiler.

3 To implement your functions, use this as an example for how to implement a receive function (RS232 implementation):

```
/* Receive Interrupt routine for RealLink */
#if (VS_REALLINKMODE == 1)
static void UART1Interrupt()
{
  switch(U1FCR_bit.IID)
  {
  case IIR_CTI:
  case IIR_RDA:                  /* Receive data available */
    VS_RealLinkReceive(U1RBR);   /* Call received byte callback */
                                 /* function */
    break;
  case IIR_THRE:           /* THRE interrupt */
  case 0x0:                /* Modem interrupt */
  case IIR_RSL:          /* Receive line status interrupt (RDA) */
               /* Character timeout indicator interrupt (CTI) */
  default:
    break;
  }
 VICVectAddr = 0;
}
#endif
```

4 Include `RealLink.h` in the file where the `Transmit()` and `Reset()` functions are implemented.

Step 5: To complete the target source code

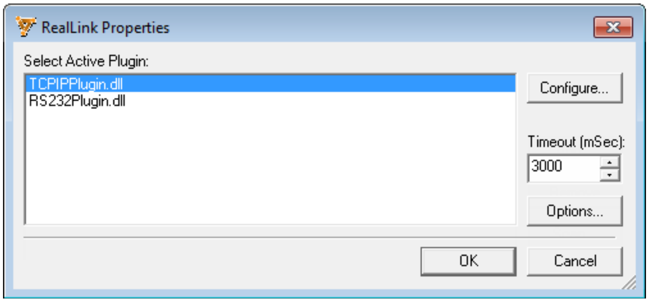
1 Compile and link the complete project.

2 Download the source code to the target.

Step 6: To configure the Validator for RealLink

1 Start the Validator and open your workspace.

2 Choose **RealLink>Properties** to open the **RealLink Properties** dialog box.



3 In the **Select Active Plugin** list, select which communication plugin module to use.

4 To configure the communication plugin module with the same settings as those implemented on the target, click the **Configure** button. A dialog box is displayed. For more information about the settings, see:

- *RealLink TCP/IP Communication Setup dialog box*, page 807
- *RealLink RS232 Communication Setup dialog box*, page 808

Information about the selected RealLink communication plugin is stored in the current Validator workspace.

5 If you are using TCP/IP, you might find it useful to add the `RL_TCPIP.cpp` file—which you can find in the `Examples\SampleCode` directory in your product installation—to your target project. This 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.

Alternatively, if you prefer to set up your own TCP/IP communication on the target instead of using `RL_TCPIP.cpp`:

- 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.
- Each time a connection is established on this port, extract the Validator IP address from the connection.
- 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.

- 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.

6 When finished, continue with *Establishing the first RealLink connection*, page 799.

ESTABLISHING THE FIRST REALLINK CONNECTION

When the communication plugin module has been configured, you can establish a RealLink connection.

1 Choose **RealLink>Connect**.

2 If the connection is successfully established, the Validator **Output** window displays a message about it.

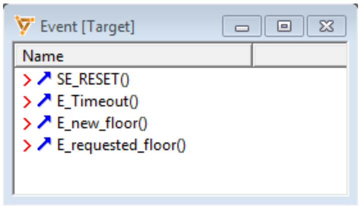
3 When the RealLink connection has been successfully established, the Validator stops the execution when the `VS_WAIT()` macro is reached for the first time (`VS_WAIT()` is the macro that you inserted in the target application code). `VS_WAIT()` continuously checks whether execution should be halted.

You can now monitor and control the target application.

CHANGING BETWEEN VALIDATOR MODE AND TARGET MODE

1 In the Validator, select the window you want to change values for. For example, the **Events** window.

2 Right-click and choose **Show target values** to select or deselect showing values as they are on target (or press Alt + F8).

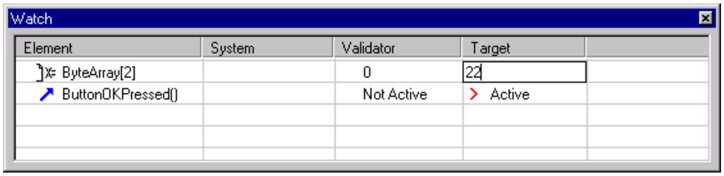


The window title reflects that the values are based on real target values.

CHANGING VARIABLE VALUES ON TARGET

When the VS_WAIT macro is reached and execution of your target application stops, you can change the value of a variable.

1 To change the value of a variable, use either the **Variables** window or the **Watch** window:



2 Type the new value in the value field.

SENDING EVENTS TO TARGET

When the VS_WAIT macro is reached and execution of your target application stops, you can send events to the target.

1 In the **Events** window (or select an event in the **Watch** window and press Enter), double-click an event.

The event will be sent to 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 on the target.

2 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 these ways:

- If an event that occurred on target is processed and the event is shown either in the **Events** 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.

Note: In Autostep mode and Run mode, you cannot send events to the state machine model that is running on the target.

CONTROLLING APPLICATION EXECUTION ON TARGET

You can break execution of code on target. Breaks are performed on these two macros:

- `VS_WAIT`, which you must insert manually in the `main` loop; see *Setting up RealLink*, page 791. When `VS_WAIT` is reached, the Validator exchanges data with the runtime application and updates all logical elements, according to the options selected. A break on this macro corresponds to a break on a macrostep.
- A macro in the Visual State API which is parallel to `VS_WAIT`. Break on the API macro corresponds to a break on a microstep.

For information about macrosteps and microsteps, see *Runtime behavior—macrosteps and microsteps*, page 122.

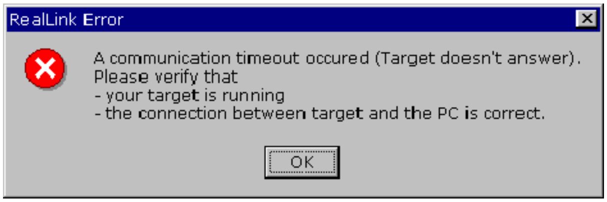
1 Immediately after the RealLink connection with the target has been established, the Validator will try to stop execution of the code when the first instance of the `VS_WAIT` macro is reached. When code execution stops, you can use the RealLink menu commands to continue execution and thereby debug your application.

For information about the commands, see *RealLink menu*, page 804.

2 Continue using the commands on the menu until you are ready.

TROUBLESHOOTING

If RealLink fails to connect to the target microcontroller, a message box appears (the message depends on the specific error):



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 issues

- Does the implementation of the main loop follow the sample code that you can find in step 3, *Setting up RealLink*, page 791?
- Is the cable between the host computer and the target microcontroller connected?
- Is the target microcontroller powered on?

- Have you generated code from the Coder with RealLink enabled and have you downloaded the compiled code to the target?
- Is the correct communication plugin module selected? For information about how to configure the Validator for RealLink, see *Setting up RealLink*, page 791.
- Is the communication plugin correctly configured—does it match the target settings? See *Setting up RealLink*, page 791.
- Is the cable between the host computer 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?

Use a terminal program to transmit a known value to the microcontroller and have it echo it back. For example, use a program such as HyperTerminal, which might be found on the Internet and which used to be shipped with older versions of Microsoft Windows.

Settings for the 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.
- 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.

Version control

- Is the state machine model loaded in the Validator the same as the one running in the target microcontroller? If not, load the correct diagram into the Validator.
- Have you changed the state machine model after you compiled and downloaded the Coder-generated files to the target? In this case, code-generate your state machine model again, build the complete target application, and download it to the microcontroller.

RealLink memory consumption

Using RealLink will increase the size of the generated code. Memory consumption depends on:

- State machine model dependent memory use
- RealLink API dependent memory use

STATE MACHINE MODEL DEPENDENT MEMORY USE

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 IAR Visual State.

The exact memory usage in bytes for CONST DATA memory and DATA can be found by means of below formulas based on these constituents:

S =	Number of Visual State 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 data types

Items in `monospace` font refer to code generated by the Coder.

Memory use in bytes for each Visual State project

CONST DATA = (10 + S) * CDP + (1 + GEV) * DP + 10 * ST +13

Memory use in bytes for each Visual State system

CONST DATA =
8 * CDP + FP + (2 + GEV) * DP + (AE + 1) * AET + EP * ST + (IVT + 1) * ST

Additional memory usage due to code generation with Uniform API

Code generated by the Visual State Coder for the Uniform API requires additional memory use which is calculated as follows:

`DATA` = S * size of SEM_CONTEXT pointer

REALLINK API DEPENDENT MEMORY USE

The RealLink API memory use largely depends on the compiler you are using.

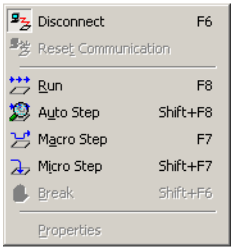
Graphical environment for RealLink

Reference information about:

- *RealLink menu*, page 804
- *RealLink Properties dialog box*, page 806
- *RealLink TCP/IP Communication Setup dialog box*, page 807
- *RealLink RS232 Communication Setup dialog box*, page 808
- *RealLink Options dialog box*, page 809

RealLink menu

The **RealLink** menu provides commands for debugging using RealLink:



Menu commands

These commands are available on the menu:

Connect/Disconnect

Connects or disconnects to the target board.

Reset Communication

Resets the communication with the target board.

Run

Executes as fast as possible. 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 `VS_WAIT`, the target checks whether or not it should stop execution. Note that if **Debug>Record** is used, Run mode corresponds to Autostep mode because the values of all Visual State elements are needed for the test sequence file.

Auto Step

Executes the code on target, while at the same time monitoring the values of the Visual State elements. Each time a microstep or macrostep is reached, the values of the elements are updated. When the values have been updated, the execution in target continues.

Macro Step

Executes until the `VS_WAIT` macro is reached. The behavior depends on whether the starting point is that execution stops on a microstep or a macrostep:

Starting point: microstep (the microstep macro)—which 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 `VS_WAIT` macro is reached.

Starting point: macrostep (the `VS_WAIT` macro)—which means that 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 `VS_WAIT` macro is reached again. As with the microstep, if there are no events in the queue, this corresponds to one loop in the Visual State main loop, without any processing being performed.

See *Runtime behavior—macrosteps and microsteps*, page 122.

Micro Step

Performs a deduction with the next trigger. In other words, execution continues until either the `VS_WAIT` macro or the parallel microstep macro in the Visual State API is reached. The behavior depends on whether the starting point is that execution stops on a microstep or a macrostep:

Starting point: microstep (the microstep macro)—which means that there are signals in the signal queue. Thus, a deduction will be performed using the first signal in the queue.

Starting point: macrostep (the `VS_WAIT` macro)—which means that a deduction is 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 Visual State 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.

See *Runtime behavior—macrosteps and microsteps*, page 122.

Break

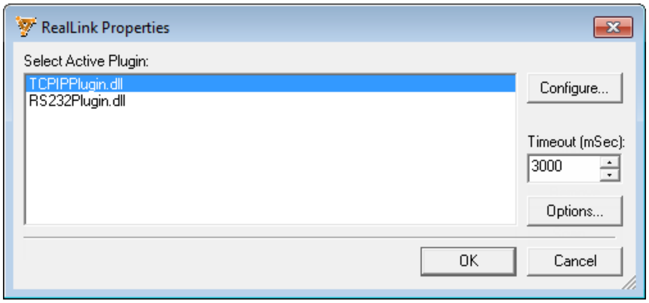
Breaks the execution.

Properties

Displays the **RealLink Properties** dialog box, see *RealLink Properties dialog box*, page 806.

RealLink Properties dialog box

The **RealLink Properties** dialog box is available from the **RealLink** menu.



Use this dialog box to configure the RealLink connection.

Select Active Plugin

Select the communication plugin that you are going to use.

Configure

Displays the **RealLink TCP/IP Communication Setup** dialog box or the **RealLink RS232 Communication Setup** dialog box, depending on which plugin you have selected in the list. See *RealLink TCP/IP Communication Setup dialog box*, page 807 and *RealLink RS232 Communication Setup dialog box*, page 808, respectively.

Timeout

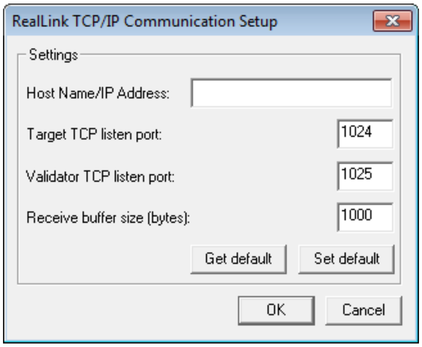
Specify the number of milliseconds that the Validator waits for a response from the target board before timing out.

Options

Displays the **RealLink Options** dialog box, see *RealLink Options dialog box*, page 809.

RealLink TCP/IP Communication Setup dialog box

The **RealLink TCP/IP Communication Setup** dialog box is available from the **RealLink Properties** dialog box.



Use this dialog box to configure TCP/IP communication with a target board.

Host Name/IP Address

Type the target host name or IP address.

Target TCP listen port

Specify the target listen port.

The reason for this is that both the target and the RealLink TCP/IP communication plugin listen on a specific port to establish a connection to the target. By default, these ports are used:

- Port 1024 is used as the target listen port.
- Port 1025 is used as the Validator listen port.

Validator TCP listen port

Specify the Validator listen port.

Receive buffer size

Specify the size of the receive buffer.

The suitable size depends on your state machine model. Set the 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.

Get default

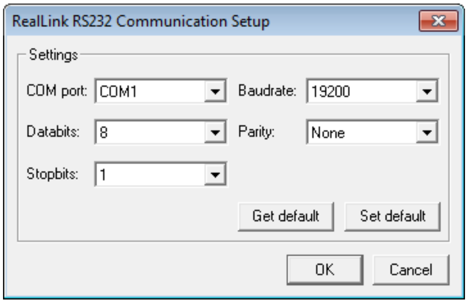
Restores the TCP/IP communication settings to the default values.

Set default

Saves the current TCP/IP communication settings as the new default values.

RealLink RS232 Communication Setup dialog box

The **RealLink RS232 Communication Setup** dialog box is available from the **RealLink Properties** dialog box.



Use this dialog box to configure RS232 communication with a target board.

Note: The Visual State 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.

COM port

Select one of the supported communication ports: `COM1`, `COM2`, `COM3`, or `COM4`.

Databits

Select the number of data bits: 6, 7, or 8.

Stopbits

Select the number of stop bits: 1, 1 ½, or 2.

Baudrate

Select one of these communication speeds: 2400, 9600, 19200, 38400, 57600, or 115200.

Parity

Select the parity: `None`, `Odd`, `Even`, `Mark`, or `Space`.

Get default

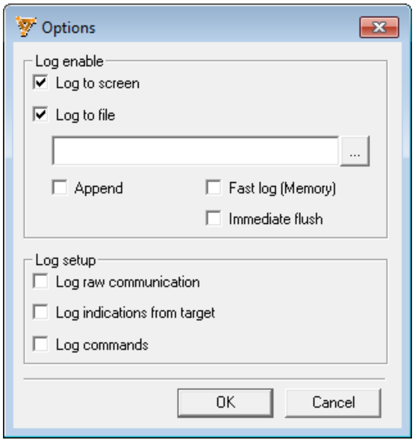
Restores the RS232 communication settings to the default values.

Set default

Saves the current RS232 communication settings as the new default values.

RealLink Options dialog box

The **RealLink Options** dialog box is available from the **RealLink Properties** dialog box.



Use this dialog box to configure RealLink logging.

Log to screen

Directs a log of the RealLink communication to the **Validator** page of the **Output** window.

Log to file

Saves a log of the RealLink communication to the text file that you specify in the text box. A browse button is available for your convenience.

Append

Appends all newly logged information at the end of the existing log without overwriting the old text.

Fast log (Memory)

The logging will be done to memory. When the connection is closed, the actual logging to the file will take place.

Immediate flush

The communication will be logged to the file. If this is selected, the data will be flushed to the log file on the disk every time there is something to report. If this is not selected, the data will be written to the file on the disk at the discretion of the file system.

Log raw communication

Logs all communications exactly as it is transmitted. This format requires specialized knowledge to interpret.

Log indications from target

Logs the indications from the target without logging all the data that might be related to the indications. To interpret this format, you need specialized knowledge.

Log commands

Logs just the commands sent to the target board.

Part 8. Documenting Visual State projects using the Documenter

This part of the *IAR Visual State User Guide* includes these chapters:

- Documenting projects
- Documenter command line options

Documenting projects

- Introduction to documenting projects using the Documenter
- Creating project reports using the Documenter
- Graphical environment for the Documenter

Introduction to documenting projects using the Documenter

Learn more about:

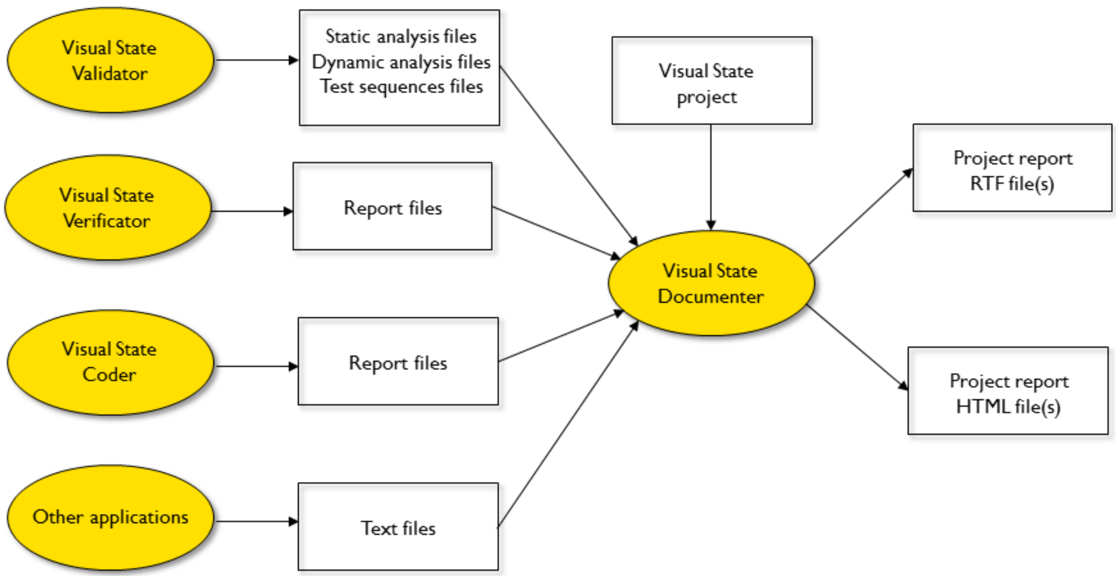
- *A project report*, page 813

A PROJECT REPORT

For documentation of your Visual State projects, you can create customized reports by using the Visual State Documenter. The Documenter can be activated via the Navigator or the command line.

A project report generated by the Documenter includes information on design, functional and formal testing, generated code and implementation of your project. All relevant project information is collected from the other Visual State components and organized into a structured document. The document can be in HTML format, or RTF (rich text format), according to your choice.

The information in the project report is based on a number of Visual State files, as can be seen in this figure:



You can specify which information should be included in the report, for example design and test, just as you can also choose between various levels of details for the report. See *Creating a project report*, page 814.

Creating project reports using the Documenter

Read about:

- *Creating a project report*, page 814

CREATING A PROJECT REPORT

1 Start the Navigator and open your workspace file.

2 In the **Workspace Browser** window, select the project for which to create a report. Right-click and choose **Options>Documentation**.

The **Documenter Options** dialog box is displayed. For reference information, see *Documenter Options dialog box*, page 816.

Make your settings and click **OK**.

3 Choose **Project>Document** to start the report generation. Progress information is listed in the **Output** window.

The generated report is displayed in the HTML viewer of the Navigator, and a `reports` directory for the report is created in the browser. The generated project report (filename extension `rtf`) is located in the `Doc` subdirectory in the directory that contains your Visual State project file.

Note: If you have opened a generated project report in Microsoft Word, close the file before you start creating a new project report in RTF. For some systems it might also be necessary to close the Microsoft Word application. Also, you will probably find that the table of contents is not updated. To update it, right-click the table of contents and choose **Update Field** from the context menu. To update the page references in the entire document, press Ctrl+A to select all and press F9 to update all fields.

To change settings for the project report, see *Documenter Options dialog box*, page 816.

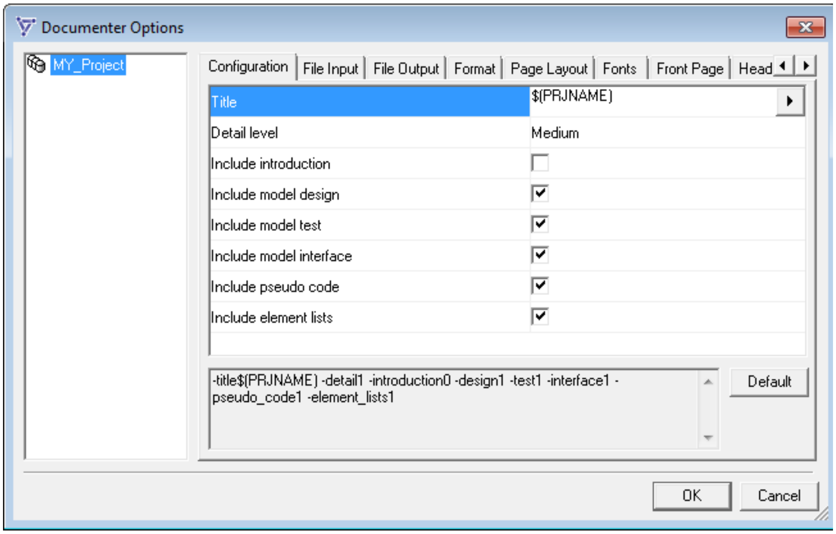
Graphical environment for the Documenter

Reference information about:

- *Documenter Options dialog box*, page 816

Documenter Options dialog box

The **Documenter Options** dialog box is available from the **Project** menu in the Navigator.



Use this dialog box to set options for generating documentation reports for your project. All options are set on project level.

For a description of an option, right-click it or select it and press Shift+F1.

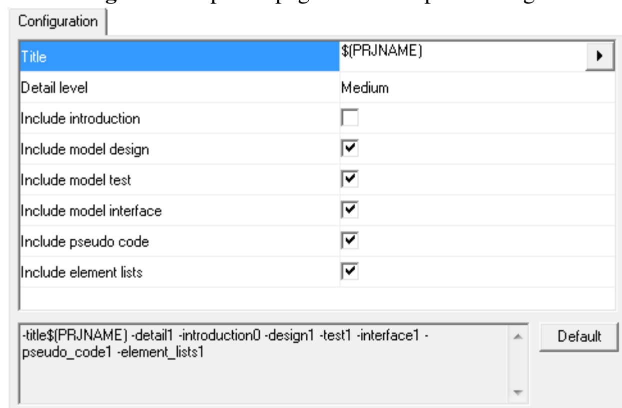
You can set options on these tabbed pages:

- *Documenter Options dialog box : Configuration*, page 817
- *Documenter Options dialog box : File Input*, page 819
- *Documenter Options dialog box : File Output*, page 821
- *Documenter Options dialog box : Format*, page 823
- *Documenter Options dialog box : Page Layout*, page 824
- *Documenter Options dialog box : Fonts*, page 826
- *Documenter Options dialog box : Front Page*, page 827
- *Documenter Options dialog box : Header/Footer*, page 829
- *Documenter Options dialog box : RTF Styles*, page 831
- *Documenter Options dialog box : HTML Styles*, page 834

See also *Creating a project report*, page 814.

Documenter Options dialog box : Configuration

The **Configuration** options page contains options for general configuration.



Use this page to specify the name of the report, which sections in the report to be included, and the detail level of the report. The display area under the options shows the resulting command line for the report generation.

Title

Specify the title of the report.

Detail level

Select the detail level of the report.

Choose between:

Low

Comments, state vectors from Validator test sequence files, and transitions and reactions are excluded from the report.

Medium

Comments and state vectors from Validator test sequence files are excluded from the report.

High

All information related to a project is included in the report.

Include introduction

Includes an introduction in the report, consisting of user-written text files.

Include model design

Includes information on your state machine model in the report. This is the main section of the report. It contains a complete description of the model, including diagrams, transitions, elements, etc.

Include model test

Includes information from your testing in the report. This section contains test files such as Validator static analysis files, Validator dynamic analysis files, Validator test sequence files, and Verificator report files.

Include model interface

Includes information on the interface of your design in the report. This section contains a table for each transition element type that is part of the external interface: action functions, external variables, and constants.

Include pseudo code

Includes pseudo code for the project in the report.

Include element lists

Includes transition element lists in the report. This section contains a table for each transition element type: events, event groups, action functions, external variables, internal variables, signals, constants, enumerators, and external states.

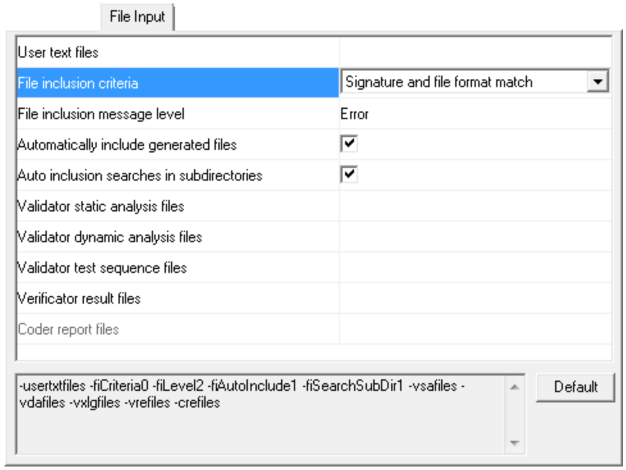
External states are declarations of states defined in another `vsr` file. The declarations are created automatically by the Designer when states in another `vsr` file are referenced, for example when using state conditions for a state in another `vsr` file.

Default

Restores the options to their default settings.

Documenter Options dialog box : File Input

The **File Input** options page contains options for file input to the Documenter.



Use this page to specify the files to be used as input for your project report. The display area under the options shows the resulting command line for the report generation.

To ensure consistency between the Visual State generated files to be used as input for the report and the Visual State project, the files are checked. By default, the generated files are only included in the report if their digital signatures correspond to the digital signature of the loaded project.

User text files

Specify paths to user text files to include in the introduction section of the report.

File inclusion criteria

Controls the criteria for files to be included in the project report. Only files meeting the file inclusion criteria will be included. Choose between:

Signature and file format match

The signature (thus, also the name of the project file) and the file format must all match.

Project filename and format match

The signatures do not need to match, but the name of the project file and file format must match.

File format match

The signatures and the name of the project file do not need to match, but the file format must match.

None

No criteria is used for determining which files to include.

File inclusion message level

Select the message level to use if an included file does not meet the criteria for inclusion of generated files.

Choose between:

Information

A message will inform you if an included file does not meet the criteria for inclusion of generated files.

Warning

A warning will be generated if an included file does not meet the criteria for inclusion of generated files.

Error

An error will be generated if an included file does not meet the criteria for inclusion of generated files.

Automatically include generated files

Automatically include all generated files that contain a digital signature, such as Validator test sequence files, Coder result files, etc. Only files meeting the file inclusion criteria will be included.

Auto inclusion searches in subdirectories

Includes generated files in subdirectories relative to the location of the project file in the report.

Validator static analysis files

Specify paths to the Validator static analysis files to include in the report.

Validator dynamic analysis files

Specify paths to the Validator dynamic analysis files to include in the report.

Validator test sequence files

Specify paths to the Validator test sequence files to include in the report.

Verificator result files

Specify paths to the Verificator result files to include in the report.

Coder report files

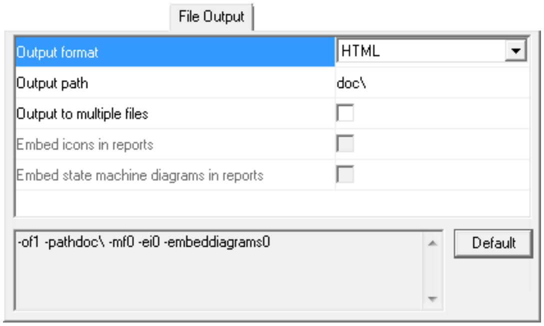
Specify paths to the Coder report files to include in the report.

Default

Restores the options to their default settings.

Documenter Options dialog box : File Output

The **File Output** options page contains options for file output from the Documenter.



Use this page to make file output settings for your project report. The display area under the options shows the resulting command line for the report generation.

Output format

Select the output format for the report.

Choose between:

RTF

Creates a report in RTF (Rich Text Format) format.

The generated RTF output conforms to the RTF specification, version 1.6, except for these Documenter-specific fields:

`REF`:	Used for inserting links to bookmarks.
`PAGEREF`:	Used for inserting links to pages.

`INCLUDEPICTURE`:	Used for inserting links to image files (icons and state machine diagrams).
`TOC`:	Used for inserting a table of contents.

HTML

Creates a report in HTML format. In addition, a single CSS2 file is generated. The styles of the CSS2 file are based on the option that you specify on the **Page Layout** page.

All images, such as icons and state machine diagrams are generated in separate files that are linked to the HTML output. Note that the diagrams are generated in EMF format, which is non-standard HTML. Thus, diagrams in output might not be available in all web browsers.

The generated HTML output generally conforms to the HTML 4.01 Specification and the Cascading Style Sheets level 2, CSS2 Specification by W3C.

Output path

Specify the output path for all generated files. If the path does not exist, it is created automatically. The path may be a relative path.

Output to multiple files

Generates the report as a separate file for each section instead of as one single file.

Embed icons in reports

Embeds icons (as images) in the generated RTF format report. The report might grow quite large if you select this option.

If this option is deselected, all icons are generated as separate files and imported by reference (linking) in the generated RTF format report. This violates the RTF standard and the resulting file might not be readable by all word processors.

Embed state machine diagrams in reports

Embeds state machine diagrams (as images) in the generated RTF format report.

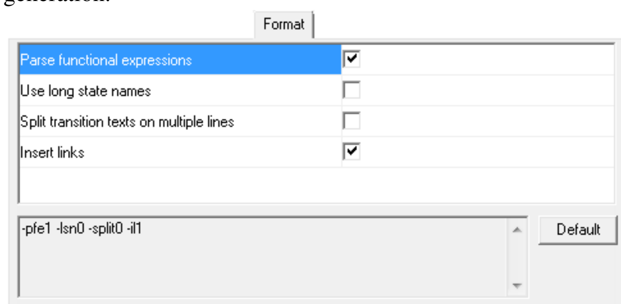
If this option is deselected, all images of state machine diagrams are generated as separate files and imported by reference (linking) in the generated RTF format report. This violates the RTF standard and the resulting file might not be readable by all word processors.

Default

Restores the options to their default settings.

Documenter Options dialog box : Format

The **Format** options page contains some formatting options for the Documenter report generation.



Use this page to make formatting settings for the Documenter. The display area under the options shows the resulting command line for the report generation.

Parse functional expressions

Generates links from transition elements used in functional expressions to their respective definitions. Use this option when you generate documentation for incomplete designs that contain invalid functional expressions.

Use long state names

Uses long state names in state references.

Split transition texts on multiple lines

Divides transition texts into multiple lines in the report.

Insert links

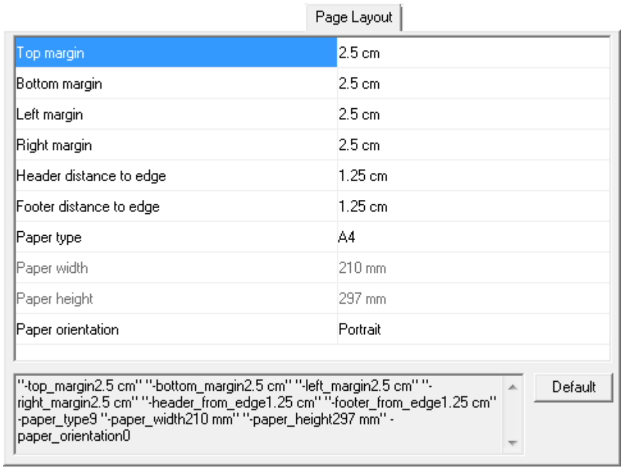
Inserts links between uses of transition elements and their associated definitions.

Default

Restores the options to their default settings.

Documenter Options dialog box : Page Layout

The **Page Layout** options page contains options for the graphical layout of the Documenter report pages.



Use this page to customize the page layout of the project report, such as margins, paper width, and paper orientation. The display area under the options shows the resulting command line for the report generation.

Top, Bottom, Left, Right margin

Specify the top, bottom, left, right margin, respectively, for the report file. The possible units are mm, cm, twips, and points.

Header distance to edge

Specify the distance from the header to the top of the page. The possible units are mm, cm, twips, and points.

Footer distance to edge

Specify the distance from the footer to the bottom of the page. The possible units are mm, cm, twips, and points.

Paper type

Select the paper size of the generated report. If you choose **User-defined**, the paper size is defined by the options **Paper width** and **Paper height**.

Paper width

Specify the width of the report page. The possible units are mm, cm, twips, and points.

Paper height

Specify the height of the report page. The possible units are mm, cm, twips, and points.

Paper orientation

Specify the orientation of the report page.

Choose between:

Portrait

The page orientation is portrait.

Landscape

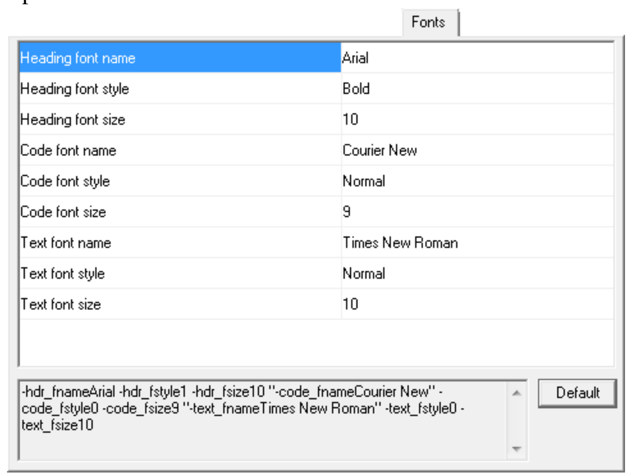
The page orientation is landscape.

Default

Restores the options to their default settings. The default settings depend on the measurement system specified for your host computer in **Regional Options** in the Control Panel.

Documenter Options dialog box : Fonts

The **Fonts** options page contains options for font use in the generated Documenter report.



Use this page to make fonts settings for the Documenter. The display area under the options shows the resulting command line for the report generation.

Heading font name

Specify the name of the font used for heading text (including text on the front page). This must exactly match the name of one of your installed fonts.

Heading font style

Select the weight of the font used for heading text (including text on the front page).

Choose between **Normal**, **Bold**, **Italic**, or **Bold Italic**.

Heading font size

Specify the size in points of the font used for heading text (including text on the front page).

Code font name

Specify the name of the font used for code (for example pseudo code). This must exactly match the name of one of your installed fonts.

Code font style

Select the weight of the font used for code (for example pseudo code).

Choose between **Normal**, **Bold**, **Italic**, or **Bold Italic**.

Code font size

Specify the size in points of the font used for code (for example pseudo code).

Text font name

Specify the name of the font used for all other text than headings and code. This must exactly match the name of one of your installed fonts.

Text font style

Select the weight of the font used for all other text than headings and code.

Choose between **Normal**, **Bold**, **Italic**, or **Bold Italic**.

Text font size

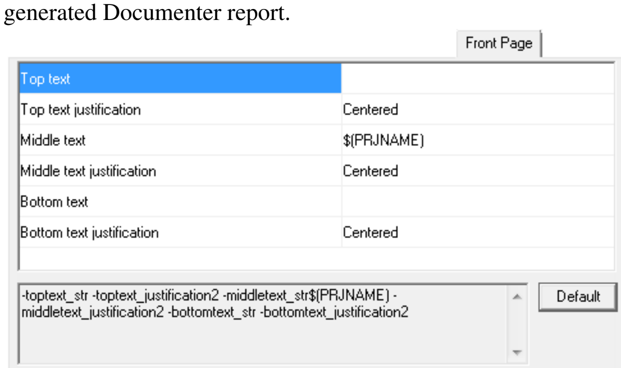
Specify the font size used for all other text than headings and code.

Default

Restores the options to their default settings.

Documenter Options dialog box : Front Page

The **Front Page** options page contains options for designing the front page of the generated Documenter report.



Use this page to make front page settings for the Documenter. The display area under the options shows the resulting command line for the report generation.

Top text

Type the text to appear at the top of the front page of a report in RTF format.

Top text justification

Select the alignment of the topmost text of the front page of a report in RTF format.

Choose between **Left**, **Centered**, or **Right**.

Middle text

Type the text to appear in the middle of the front page of a report in RTF format.

Middle text justification

Select the alignment of the text in the middle of the front page of a report in RTF format.

Choose between **Left**, **Centered**, or **Right**.

Bottom text

Type the text to appear at the bottom of the front page of a report in RTF format.

Bottom text justification

Select the alignment of text at the bottom of the front page of a report in RTF format.

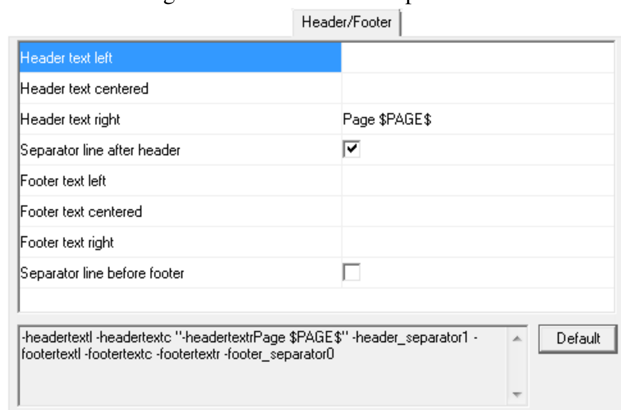
Choose between **Left**, **Centered**, or **Right**.

Default

Restores the options to their default settings.

Documenter Options dialog box : Header/Footer

The **Header/Footer** options page contains options for the appearance of the header and the footer of the generated Documenter report.



Use this page to make settings for the header and footer for the pages after the front page in the report. The display area under the options shows the resulting command line for the report generation.

Note: These options can only be set for the RTF output format.

Header text left

Type the text string to appear at the top left of the report pages.

Header text centered

Type the text string to appear in the top middle of the report pages.

Header text right

Type the text string to appear at the top right of the report pages.

Separator line after header

Prints a separator line between the page header and the body text.

Footer text left

Type the text string to appear at the bottom left of the report pages.

Footer text centered

Type the text string to appear in the bottom middle of the report pages.

Footer text right

Type the text string to appear at the bottom right of the report pages.

Separator line before footer

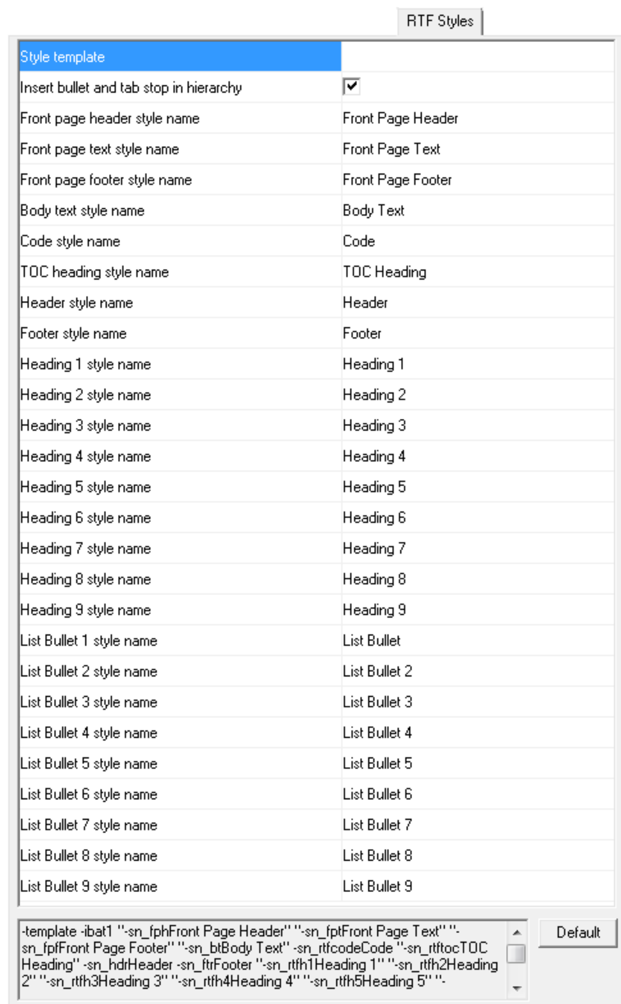
Prints a separator line between the body text and the page footer.

Default

Restores the options to their default settings.

Documenter Options dialog box : RTF Styles

The **RTF Styles** options page contains options for generating Documenter reports in RTF format.



Use this page to make your own styles and templates for a generated report in RTF. The display area under the options shows the resulting command line for the report generation.

Note: These options require that you are familiar with styles and templates in Microsoft Word or a similar program.

Style template

Specify the path to the style template used by RTF reports.

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`, make sure to 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.

Insert bullet and tab stop in hierarchy

Inserts a bullet and a tab stop in list hierarchies in RTF format reports. Deselect this option if the generated report uses an external template with list styles that by definition include such a list marker and indentation.

Front page header style name

Type the name of the front page header style in RTF format reports. The actual properties of this style are defined by other options.

Front page text style name

Type the name of the main text style of the front page in RTF format reports. The actual properties of this style are defined by other options.

Front page footer style name

Type the name of the front page footer style in RTF format reports. The actual properties of this style are defined by other options.

Body text style name

Type the name of the body text style in RTF format reports. The actual properties of this style are defined by other options.

Code style name

Type the name of the code style in RTF format reports. The actual properties of this style are defined by other options.

TOC heading style name

Type the name of the heading style of the table of contents of RTF format reports. The actual properties of this style are defined by other options.

Header style name

Type the name of the header style in RTF format reports. The actual properties of this style are defined by other options.

Footer style name

Type the name of the footer style in RTF format reports. The actual properties of this style are defined by other options.

Heading # style name

Type the name of the style for top-level headings in RTF format reports. The actual properties of this style are defined by other options.

List Bullet # style name

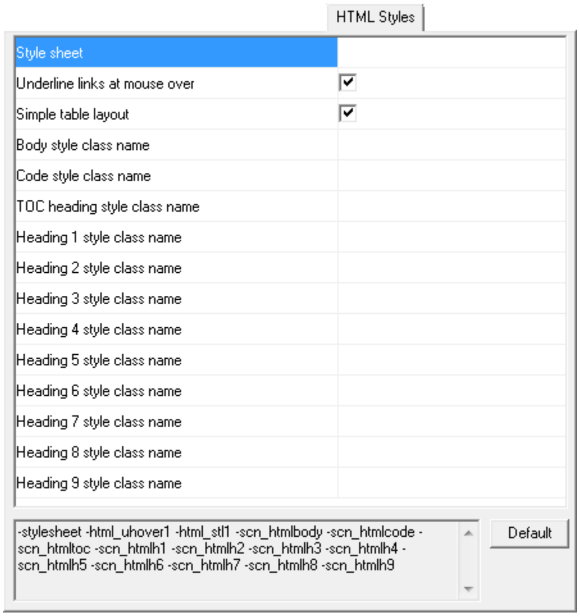
Type the name of the style for top-level list bullets in RTF format reports. The actual properties of this style are defined by other options.

Default

Restores the options to their default settings.

Documenter Options dialog box : HTML Styles

The **HTML Styles** options page contains options for generating Documenter reports in HTML format.



Use this page to make your own styles and style sheets for the generated report in HTML format. The display area under the options shows the resulting command line for the report generation.

Note: These options require that you are familiar with styles and style sheets in HTML and CSS2.

Style sheet

Specify the path to the CSS style sheet used by HTML reports.

Underline links at mouse over

Makes hypertext links underlined only when the mouse pointer hovers over the link.

Simple table layout

Uses a simplified layout for tables.

Body style class name

Type the name for the body style class (the HTML element `body`). The actual properties of this style are defined by other options.

Code style class name

Type the name for the code style class (the HTML element `pre`). The actual properties of this style are defined by other options.

TOC heading style class name

Type the name for the heading style class for the table of contents (the HTML element `h1`). The actual properties of this style are defined by other options.

Heading # style class name

Type the name for the top-level heading style class (the HTML element `h1`). The actual properties of this style are defined by other options.

Default

Restores the options to their default settings.

Documenter command line options

- Introduction to invoking the Documenter using command line options
- Summary of Documenter options
- Descriptions of Documenter options.

Introduction to invoking the Documenter using command line options

Learn more about:

- *Briefly about invoking the Documenter*, page 837
- *Invocation syntax for the Documenter*, page 837

BRIEFLY ABOUT INVOKING THE DOCUMENTER

You can set Documenter options either in the Navigator—using the **Documenter Options** dialog box—or via the command line. For each option available in the **Documenter Options** dialog box, there is an equivalent option for the command line.

INVOCATION SYNTAX FOR THE DOCUMENTER

This is the invocation syntax for starting the Documenter from the command line:

```
Documenter.exe Vsp_file [--l] [--@filename]-option[argument]*
```

Where:

`--l`	Loads options from the `vtg` file that corresponds to the specified `vsp` file.
`--@`	Loads 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 `//`.

Summary of Documenter options

This table summarizes the Documenter command line options:

Command line option	Description
-bottom_margin	Sets the bottom margin for the report file.
-bottomtext_justification	Determines the alignment of the text at the bottom of the front page of an RTF report.
-bottomtext_str	Determines the text at the bottom of the front page of an RTF report.
-code_fname	Determines the font used for code.
-code_fsize	Determines the font size used for code.
-code_fstyle	Determines the weight of the font used for code.
-design	Includes/excludes information on the state machine in the report.
-detail	Determines the detail level of the report.
-ei	Enables/disables embedding icon images in the generated RTF report.
-element_lists	Includes/excludes transition element lists from the report.
-embeddiagrams	Enables/disables embedding state machine diagrams images in the generated RTF report.
-fiAutoInclude	Enables/disables automatic inclusion of all generated files that contain a digital signature.
-fiCriteria	Determines the criteria for inclusion of generated files that contain a digital signature.
-fiLevel	Determines the message level to use if an included file does not meet the criteria for inclusion.
-fiSearchSubDir	Includes/excludes generated files in subdirectories relative to the project file.
-footer_from_edge	Sets the distance from the footer to the bottom of the page.
-footer_separator	Enables/disables printing a separator line between the body text and the page footer.
-footertextc	Specifies the text string in the bottom middle of the report pages.
-footertextl	Specifies the text string at the bottom left of the report pages.
-footertextr	Specifies the text string at the bottom right of the report pages.
-fullstatenames	Enables/disables long state names in state references.
-hdr_fname	Determines the font used for heading text.

Table 39: Documenter command line options

Command line option	Description
-hdr_fsize	Determines the font size used for heading text.
-hdr_fstyle	Determines the weight of the font used for heading text.
-header_from_edge	Specifies the distance from the header to the top of the page.
-header_separator	Enables/disables printing a separator line between the body text and the page header.
-headertextc	Specifies the text string in the top middle of the report pages.
-headertextl	Specifies the text string at the top left of the report pages.
-headertextr	Specifies the text string at the top right of the report pages.
-html_stl	Enables/disables a simplified layout for tables.
-html_uhover	Sets how hypertext links are underlined in an HTML report.
-ibat	Enables/disables insertion of a bullet and a tab stop in list hierarchies.
-il	Enables/disables insertion of links between uses of transition elements and their associated definitions.
-interface	Includes information on the interface of the design in the report.
-introduction	Includes an introduction consisting of user-written text files.
-left_margin	Sets the left margin for the report file.
-mf	Determines whether to generate the report as one single file or as a separate file for each section.
-middletext_justification	Determines the alignment of the text in the middle of the front page of an RTF report.
-middletext_str	Determines the text in the middle of the front page of an RTF report.
-of	Toggles the output format for the report between HTML and RTF.
-paper_height	Determines the height of the report page.
-paper_orientation	Determines the orientation of the report page.
-paper_type	Sets the paper size of the generated report.
-paper_width	Determines the width of the report page.
-path	Specifies the output path for all generated files.
-pfe	Enables/disables links from transition elements used in functional expressions to their respective definitions.
-pseudo_code	Includes/excludes pseudo code for the project in the report.
-right_margin	Sets the right margin for the report file.
-scn_htmlbody	Specifies a name for the HTML body style class.

Table 39: Documenter command line options

Command line option	Description
-scn_htmlcode	Specifies a name for the HTML code style class.
-scn_html1	Specifies a name for the HTML top-level heading style class.
-scn_html2	Specifies a name for the HTML level 2 heading style class.
-scn_html3	Specifies a name for the HTML level 3 heading style class.
-scn_html4	Specifies a name for the HTML level 4 heading style class.
-scn_html5	Specifies a name for the HTML level 5 heading style class.
-scn_html6	Specifies a name for the HTML level 6 heading style class.
-scn_html7	Specifies a name for the HTML level 7 heading style class.
-scn_html8	Specifies a name for the HTML level 8 heading style class.
-scn_html9	Specifies a name for the HTML level 9 heading style class.
-scn_htmltoc	Specifies a name for the HTML heading style class for the table of contents.
-sn_bt	Determines the name of the body text style in RTF reports.
-sn_fpf	Determines the name of the front page footer style in RTF reports.
-sn_fph	Determines the name of the front page header style in RTF reports.
-sn_fpt	Determines the name of the main text style of the front page in RTF reports.
-sn_ftr	Determines the name of the footer style in RTF reports.
-sn_hdr	Determines the name of the header style in RTF reports.
-sn_lb1	Determines the name of the style for top-level list bullets in RTF reports.
-sn_lb2	Determines the name of the style for level 2 list bullets in RTF reports.
-sn_lb3	Determines the name of the style for level 3 list bullets in RTF reports.
-sn_lb4	Determines the name of the style for level 4 list bullets in RTF reports.
-sn_lb5	Determines the name of the style for level 5 list bullets in RTF reports.
-sn_lb6	Determines the name of the style for level 6 list bullets in RTF reports.
-sn_lb7	Determines the name of the style for level 7 list bullets in RTF reports.

Table 39: Documenter command line options

Command line option	Description
-sn_lb8	Determines the name of the style for level 8 list bullets in RTF reports.
-sn_lb9	Determines the name of the style for level 9 list bullets in RTF reports.
-sn_rtfcode	Determines the name of the code style in RTF reports.
-sn_rtfh1	Determines the name of the style for top-level headings in RTF reports.
-sn_rtfh2	Determines the name of the style for level 2 headings in RTF reports.
-sn_rtfh3	Determines the name of the style for level 3 headings in RTF reports.
-sn_rtfh4	Determines the name of the style for level 4 headings in RTF reports.
-sn_rtfh5	Determines the name of the style for level 5 headings in RTF reports.
-sn_rtfh6	Determines the name of the style for level 6 headings in RTF reports.
-sn_rtfh7	Determines the name of the style for level 7 headings in RTF reports.
-sn_rtfh8	Determines the name of the style for level 8 headings in RTF reports.
-sn_rtfh9	Determines the name of the style for level 9 headings in RTF reports.
-sn_rtftoc	Determines the name of the heading style of the table of contents of RTF reports.
-split	Enables/disables dividing transition texts into multiple lines in the report.
-stylesheet	Specifies the CSS style sheet used by HTML reports.
-template	Specifies the style template used by RTF reports.
-test	Includes/excludes information from the testing in the report.
-text_fname	Determines the font used for all other text than headings and code.
-text_fsize	Determines the font size used for all other text than headings and code.
-text_fstyle	Determines the weight of the font used for all other text than headings and code.
-title	Specifies the title of the report.
-top_margin	Sets the top margin for the report file.
-toptext_justification	Determines the alignment of the topmost text of the front page of an RTF report.
-toptext_str	Determines the topmost text of the front page of an RTF report.
-usertxtfiles	Specifies which user text files to include in the report.
-variant	Specifies which variant to create a report for.

Table 39: Documenter command line options

Command line option	Description
`-vdafiles`	Specifies which Validator dynamic analysis files to include in the report.
`-vlgfiles`	Specifies which Validator test sequence files to include in the report.
`-vrefiles`	Specifies which Verificator result files to include in the report.
`-vsafiles`	Specifies which Validator static analysis files to include in the report.

Table 39: Documenter command line options

Descriptions of Documenter options

The following pages give detailed reference information about each Documenter command line option.

Note: All Documenter command line options are set on project level.

-bottom_margin

Syntax `-bottom_margin`*`size`*`{cm|mm|twips|points}`

Parameters

`size` The size of the margin in the given unit specified as `double`. By default, the size is set to `2.5cm`.

Description Sets the bottom margin for the report file.

Project>Options>Documentation>Page Layout>Bottom margin

-bottomtext_justification

Syntax `-bottomtext_justification{0|1|2}`

Parameters

`0` The text at the bottom of the front page is aligned to the left.

`1` (default) The text at the bottom of the front page is centered.

`2` The text at the bottom of the front page is aligned to the right.

Description Determines the alignment of the text at the bottom of the front page of a report in RTF format.

Project>Options>Documentation>Front Page>Bottom text justification

-bottomtext_str

Syntax `-bottomtext_str`*`text`*

Parameters

`text`	The text at the bottom of the front page.

Description Determines the text at the bottom of the front page of a report in RTF format.

Project>Options>Documentation>Front Page>Bottom text

-code_fname

Syntax `-code_fname`*`font`*

Parameters

`font`	The name of the font used for code (for example pseudo code). This must exactly match the name of one of your installed fonts. By default, the value is `Courier New`.

Description Determines the font used for code (for example pseudo code).

Project>Options>Documentation>Fonts>Code font name

-code_fsize

Syntax `-code_fsize`*`size`*

Parameters

`size`	An integer that represents the size in points of the font used for code (for example pseudo code). By default, the value is `9`.

Description Determines the font size used for code (for example pseudo code).

Project>Options>Documentation>Fonts>Code font size

-code_fstyle

Syntax `-code_fstyle{0|1|2|3}`

Parameters

`0` (default)	The code font weight is Normal.
`1`	The code font weight is Bold.
`2`	The code font weight is Italic.
`3`	The code font weight is Bold Italic.

Description

Determines the weight of the font used for code (for example pseudo code).

Project>Options>Documentation>Fonts>Code font style

-design

Syntax `-design{0|1}`

Parameters

`0`	Does not include information on your state machine in the report.
`1` (default)	Includes information on your state machine in the report.

Description

Determines whether to include information on your state machine. This is the main section of the report. It contains a complete description of the design, including diagrams, transitions, elements, etc.

Project>Options>Documentation>Configuration>Include model design

-detail

Syntax `-detail{0|1|2}`

Parameters

`0`	Low: Explanations, state vectors from Validator test sequence files, and transitions and reactions are excluded from the report.
`1` (default)	Medium: Explanations and state vectors from Validator test sequence files are excluded from the report.

`2`	High: All information related to a project is included in the report.

Description

Determines the detail level of the report.

Project>Options>Documentation>Configuration>Detail level

-ei

Syntax

`-ei{0|1}`

Parameters

`0`	Generates all icons as separate files and imports them by reference (linking) in the generated RTF format report. This violates the RTF standard and the resulting file might not be readable by all word processors.
`1` (default)	Embeds icons (as images) in the generated RTF format report. In this case, the report might grow quite large.

Description

Determines whether to embed icons (as images) in the generated RTF format report.

Project>Options>Documentation>File Output>Embed icons in report

-element_lists

Syntax

`-element_lists{0|1}`

Parameters

`0`	Does not include transition element lists in the report.
`1` (default)	Includes transition element lists in the report.

Description

Determines whether to include transition element lists. This section contains a table for each transition element type: events, event groups, action functions, external variables, internal variables, signals, constants, enumerators, and external states.

Project>Options>Documentation>Configuration>Include element lists

-embeddiagrams

Syntax `-embeddiagrams{0|1}`

Parameters

`0`	Generates all images of state machine diagrams as separate files and imports them by reference (linking) in the generated RTF format report. This violates the RTF standard and the resulting file might not be readable by all word processors.
`1` (default)	Embeds state machine diagrams (as images) in the generated RTF format report.

Description Determines whether to embed state machine diagrams (as images) in the generated RTF format report.

Project>Options>Documentation>File Output>Embed state machine diagrams in report

-fiAutoInclude

Syntax `-fiAutoInclude{0|1}`

Parameters

`0` (default)	Does not include all generated files that contain a digital signature in the report.
`1`	Includes all generated files that contain a digital signature in the report.

Description Determines whether to automatically include all generated files that contain a digital signature, such as Validator test sequence files, Coder result files, etc. Only files meeting the file inclusion criteria will be included.

See also *-fiSearchSubDir*, page 848

Project>Options>Documentation>Automatically include generated files

-fiCriteria

Syntax `-fiCriteria{0|1|2|3}`

Parameters

`0` (default)	Signature and file format match. The signatures (and thus also the project filename) and the file format must all match.
`1`	Project filename and format match. The signatures do not have to match, but the project filename and format must match.
`2`	File format match. The signatures and the project filename do not have to match, but the file format must match.
`3`	None. No criteria are used to determine which files to include.

Description Determines the criteria for inclusion of generated files that contain a digital signature, for example 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.

See also *-fiLevel*, page 847

Project>Options>Documentation>File Input>File inclusion criteria

-fiLevel

Syntax `-fiLevel{0|1|2}`

Parameters

`0`	Information. A message will inform you if an included file does not meet the criteria for inclusion of generated files.
`1`	Warning. A warning will be generated if an included file does not meet the criteria for inclusion of generated files.
`2` (default)	Error. An error will be generated if an included file does not meet the criteria for inclusion of generated files.

Description Determines the message level to use if an included file does not meet the criteria for inclusion of generated files.

See also *-fiCriteria*, page 847

Project>Options>Documentation>File Input>File inclusion message level

-fiSearchSubDir

Syntax `-fiSearchSubDir{0|1}`

Parameters

`0`	Does not include generated files in subdirectories relative to the location of the project file in the report.
`1` (default)	Includes generated files in subdirectories relative to the location of the project file in the report.

Description If you have specified the option `-fiAutoInclude1`, this option determines whether generated files in subdirectories relative to the location of the project file will also be included.

See also *-fiAutoInclude*, page 846

Project>Options>Documentation>File Input>Auto inclusion searches in subdirectories

-footer_from_edge

Syntax `-footer_from_edge`*`distance`*`{cm|mm|twips|points}`

Parameters

`distance`	The distance from the footer to the bottom of the page, in the given unit, specified as `double`. By default, set to `1.25cm`.

Description Sets the distance from the footer to the bottom of the page.

Project>Options>Documentation>Page Layout>Footer distance to edge

-footer_separator

Syntax `-footer_separator{0|1}`

Parameters

`0` (default)	Does not print a separator line between the body text and the page footer.
`1`	Prints a separator line between the body text and the page footer.

Description Determines whether to print a separator line between the body text and the page footer.

Project>Options>Documentation>Header/Footer>Separator line before footer

-footertextc

Syntax `-footertextc`*`text`*

Parameters

`text`	The centered footer text at the bottom of the report pages.

Description Specifies the text string in the bottom middle of the report pages.

Project>Options>Documentation>Header/Footer>Footer text centered

-footertextl

Syntax `-footertextl`*`text`*

Parameters

`text`	The left-aligned footer text at the bottom of the report pages.

Description Specifies the text string at the bottom left of the report pages.

Project>Options>Documentation>Header/Footer>Footer text left

-footertextr

Syntax `-footertextr`*`text`*

Parameters

text	The right-aligned footer text at the bottom of the report pages.

Description Specifies the text string at the bottom right of the report pages.

Project>Options>Documentation>Header/Footer>Footer text right

-fullstatenames

Syntax `-fullstatenames{0|1}`

Parameters

`0` (default)	Uses abbreviated state names in state references.
`1`	Uses long state names in state references.

Description Determines whether the Documenter uses long state names in state references. For example, `Tostate1.Region1.State1.Region1.State3` instead of just `State3`.

Project>Options>Documentation>Format>Use long state names

-hdr_fname

Syntax `-hdr_fname`*`font`*

Parameters

font	The name of the font used for heading text (including text on the front page). This must exactly match the name of one of your installed fonts. By default, the value is `Arial`.

Description Determines the font used for heading text (including text on the front page).

Project>Options>Documentation>Fonts>Heading font name

-hdr_fsize

Syntax `-hdr_fsize`*`size`*

Parameters

`size`	An integer that represents the size in points of the font used for heading text (including text on the front page). By default, the value is `10`.

Description Determines the font size used for heading text (including text on the front page).

Project>Options>Documentation>Fonts>Heading font size

-hdr_fstyle

Syntax `-hdr_fstyle{0|1|2|3}`

Parameters

`0`	The heading font weight is Normal.
`1` (default)	The heading font weight is Bold.
`2`	The heading font weight is Italic.
`3`	The heading font weight is Bold Italic.

Description Determines the weight of the font used for heading text (including text on the front page).

Project>Options>Documentation>Fonts>Heading font style

-header_from_edge

Syntax `-header_from_edge`*`distance`*`{cm|mm|twips|points}`

Parameters

`distance`	The distance from the header to the top of the page, in the given unit, specified as `double`. By default, set to `1.25cm`.

Description Sets the distance from the header to the top of the page.

Project>Options>Documentation>Page Layout>Header distance to edge

-header_separator

Syntax `-header_separator{0|1}`

Parameters

`0`	Does not print a separator line between the page header and the body text.
`1` (default)	Prints a separator line between the page header and the body text.

Description Determines whether to print a separator line between the page header and the body text.

Project>Options>Documentation>Header/Footer>Separator line after header

-headertextc

Syntax `-headertextc`*`text`*

Parameters

`text`	The centered header text at the top of the report pages.

Description Specifies the text string in the top middle of the report pages.

Project>Options>Documentation>Header/Footer>Header text centered

-headertextl

Syntax `-headertextl`*`text`*

Parameters

`text`	The left-aligned header text at the top of the report pages.

Description Specifies the text string at the top left of the report pages.

Project>Options>Documentation>Header/Footer>Header text left

-headertextr

Syntax `-headertextr`*`text`*

Parameters

`text`	The right-aligned header text at the top of the report pages. By default, this string is `Page` *`pagenumber`*, where *`pagenumber`* is the number of the page.

Description Specifies the text string at the top right of the report pages.

Project>Options>Documentation>Header/Footer>Header text right

-html_stl

Syntax `-html_stl{0|1}`

Parameters

`0`	Uses a textual table with no visible borders.
`1` (default)	Uses a simplified layout for tables.

Description Determines the table layout in an HTML report.

Project>Options>Documentation>HTML Styles>Simple table layout

-html_uhover

Syntax `-html_uhover{0|1}`

Parameters

`0`	Hypertext links are always underlined.
`1` (default)	Hypertext links are only underlined when the mouse pointer hovers over the link.

Description Determines how hypertext links are underlined in an HTML report.

Project>Options>Documentation>HTML Styles>Underline links at mouse over

-ibat

Syntax `-ibat{0|1}`

Parameters

`0`	Does not insert a bullet and a tab stop in list hierarchies.
`1` (default)	Inserts a bullet and a tab stop in list hierarchies.

Description

Determines whether to specifically insert a bullet and a tab stop in list hierarchies in RTF format reports. Set this option to `0` when the generated report uses an external template with list styles that by definition include such a list marker and indentation.

See also

-template, page 875

Project>Options>Documentation>RTF Styles>Insert bullet and tab stop in hierarchy

-il

Syntax `-il{0|1}`

Parameters

`0`	Does not insert links between transition elements and their associated definitions.
`1` (default)	Inserts links between transition elements and their associated definitions.

Description

Determines whether to insert links between uses of transition elements and their associated definitions.

Project>Options>Documentation>Format>Insert links

-interface

Syntax `-interface{0|1}`

Parameters

`0`	Does not include information on the interface of your design in the report.

`1` (default)	Includes information on the interface of your design in the report.

Description

Determines whether to include information on the interface of your design. This section contains a table for each transition element type that is part of the external interface: action functions, external variables, and constants.

Project>Options>Documentation>Configuration>Include model interface

-introduction

Syntax

`-introduction{0|1}`

Parameters

`0` (default)	Does not include an introduction in the report.
`1`	Includes an introduction in the report.

Description

Determines whether to include an introduction in the report, consisting of user-written text files.

See also

-*usertxtfiles*, page 878

Project>Options>Documentation>Configuration>Include introduction

-left_margin

Syntax

`-left_margin`*`size`*`{cm|mm|twips|points}`

Parameters

`size`	The size of the margin in the given unit, specified as `double`. By default, set to `2.5cm`.

Description

Sets the left margin for the report file.

Project>Options>Documentation>Page Layout>Left margin

-mf

Syntax `-mf{0|1}`

Parameters

`0` (default)	Generates the report as one single file.
`1`	Generates the report as a separate file for each section.

Description Determines whether to generate the report as one single file or as a separate file for each section.

Project>Options>Documentation>File Output>Output to multiple files

-middletext_justification

Syntax `-middletext_justification{0|1|2}`

Parameters

`0`	The text in the middle of the front page is aligned to the left.
`1` (default)	The text in the middle of the front page is centered.
`2`	The text in the middle of the front page is aligned to the right.

Description Determines the alignment of the text in the middle of the front page of a report in RTF format.

Project>Options>Documentation>Front Page>Middle text justification

-middletext_str

Syntax `-middletext_str`*`text`*

Parameters

`text`	The text in the middle of the front page. By default, this string is the name of the project.

Description Determines the text in the middle of the front page of a report in RTF format.

Project>Options>Documentation>Front Page>Middle text

-of

Syntax `-of{0|1}`

Parameters

`0` (default)	Creates a report in RTF format.
`1`	Creates a report in HTML format.

Description Specifies the output format for the report.

Project>Options>Documentation>File Output>Output format

-paper_height

Syntax `-paper_height`*`distance{cm|mm|twips|points}`*

Parameters

`distance`	The height of the page, in the given unit, specified as `double`. By default, set to `0cm`.

Description Determines the height of the report page. Use this option if you have specified the option `-paper_type0`.

See also *-paper_type*, page 858

Project>Options>Documentation>Page Layout>Paper height

-paper_orientation

Syntax `-paper_orientation{0|1}`

Parameters

`0` (default)	The page orientation is portrait.
`1`	The page orientation is landscape.

Description

Determines the orientation of the report page.

Project>Options>Documentation>Page Layout>Paper orientation

-paper_type

Syntax

`-paper_type`*`format`*

Parameters

`format` is the paper size of the generated report. Choose between:

`0`	User-defined
`1`	Letter. This is the default setting if the locale setting for the host computer uses this as the default paper format.
`2`	Letter Small
`3`	Tabloid
`4`	Ledger
`5`	Legal
`6`	Statement
`7`	Executive
`8`	A3
`9`	A4. This is the default setting if the locale setting for the host computer uses this as the default paper format.
`10`	A4 Small
`11`	A5
`12`	B4 (JIS)
`13`	B5 (JIS)
`14`	Folio
`15`	Quarto
`16`	10x14
`17`	11x17
`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

Description

Sets the paper size of the generated report. If you specify the option `-paper_type0`, you must instead specify the paper size with the options `-paper_width` and `-paper_height`.

See also

-paper_width, page 860 and *-paper_height*, page 857

Project>Options>Documentation>Page Layout>Paper type

-paper_width

Syntax `-paper_width`*`distance{cm|mm|twips|points}`*

Parameters

`distance`	The width of the page, in the given unit, specified as `double`. By default, set to `0cm`.

Description Determines the width of the report page. Use this option if you have specified the option `-paper_type0`.

See also *-paper_type*, page 858

Project>Options>Documentation>Page Layout>Paper width

-path

Syntax `-path`*`path`*

Parameters

`path`	The output path for all generated files.

Description Specifies the output path for all generated files. If the path does not exist, it is created automatically. The path can be a relative path. By default, all generated files are created in the `doc` subdirectory of the project that you generate documentation for.

Project>Options>Documentation>File Output>Output path

-pfe

Syntax `-pfe{0|1}`

Parameters

`0`	Does not generate links from transition elements in functional expressions to their respective definitions.
`1` (default)	Generates links from transition elements in functional expressions to their respective definitions.

Description Determines whether the Documenter parses functional expressions.

Project>Options>Documentation>Format>Parse functional expressions

-pseudo_code

Syntax `-pseudo_code{0|1}`

Parameters

`0`	Does not include pseudo code for the project in the report.
`1` (default)	Includes pseudo code for the project in the report.

Description Determines whether to include pseudo code for the project.

Project>Options>Documentation>Configuration>Include pseudo code

-right_margin

Syntax `-right_margin`*`size`*`{cm|mm|twips|points}`

Parameters

`size`	The size of the margin in the given unit, specified as `double`. By default, set to `2.5cm`.

Description Sets the right margin for the report file.

Project>Options>Documentation>Page Layout>Right margin

-scn_htmlbody

Syntax `-scn_htmlbody`*`name`*

Parameters

`name`	The name of the body style class.

Description Specifies a name for the body style class (the HTML element `body`). The actual properties of this class are defined by the CSS style sheet.

Project>Options>Documentation>HTML Styles>Body style class name

-scn_htmlcode

Syntax `-scn_htmlcode`*`name`*

Parameters

name	The name of the code style.

Description Specifies a name for the code style class (the HTML element `pre`). The actual properties of this class are defined by the CSS style sheet.

Project>Options>Documentation>HTML Styles>Code style class name

-scn_htmlh1

Syntax `-scn_htmlh1`*`name`*

Parameters

name	The name of the top-level heading style class.

Description Specifies a name for the top-level heading style class (the HTML element `h1`). The actual properties of this class are defined by the CSS style sheet.

Project>Options>Documentation>HTML Styles>Heading 1 style class name

-scn_htmlh2

Syntax `-scn_htmlh2`*`name`*

Parameters

name	The name of the level 2 heading style class.

Description Specifies a name for the level 2 heading style class (the HTML element `h2`). The actual properties of this class are defined by the CSS style sheet.

Project>Options>Documentation>HTML Styles>Heading 2 style class name

-scn_htmlh3

Syntax `-scn_htmlh3`*`name`*

Parameters

name The name of the level 3 heading style class.

Description Specifies a name for the level 3 heading style class (the HTML element `h3`). The actual properties of this class are defined by the CSS style sheet.

Project>Options>Documentation>HTML Styles>Heading 3 style class name

-scn_htmlh4

Syntax `-scn_htmlh4`*`name`*

Parameters

name The name of the level 4 heading style class.

Description Specifies a name for the level 4 heading style class (the HTML element `h4`). The actual properties of this class are defined by the CSS style sheet.

Project>Options>Documentation>HTML Styles>Heading 4 style class name

-scn_htmlh5

Syntax `-scn_htmlh5`*`name`*

Parameters

name The name of the level 5 heading style class.

Description Specifies a name for the level 5 heading style class (the HTML element `h5`). The actual properties of this class are defined by the CSS style sheet.

Project>Options>Documentation>HTML Styles>Heading 5 style class name

-scn_htmlh6

Syntax `-scn_htmlh6`*`name`*

Parameters

name	The name of the level 6 heading style class.

Description Specifies a name for the level 6 heading style class (the HTML element `h6`). The actual properties of this class are defined by the CSS style sheet.

Project>Options>Documentation>HTML Styles>Heading 6 style class name

-scn_htmlh7

Syntax `-scn_htmlh7`*`name`*

Parameters

name	The name of the level 7 heading style class.

Description Specifies a name for the level 7 heading style class (the HTML element `h7`). The actual properties of this class are defined by the CSS style sheet.

Project>Options>Documentation>HTML Styles>Heading 7 style class name

-scn_htmlh8

Syntax `-scn_htmlh8`*`name`*

Parameters

name	The name of the level 8 heading style class.

Description Specifies a name for the level 8 heading style class (the HTML element `h8`). The actual properties of this class are defined by the CSS style sheet.

Project>Options>Documentation>HTML Styles>Heading 8 style class name

-scn_htmlh9

Syntax `-scn_htmlh9`*`name`*

Parameters

`name` The name of the level 9 heading style class.

Description Specifies a name for the level 9 heading style class (the HTML element `h9`). The actual properties of this class are defined by the CSS style sheet.

Project>Options>Documentation>HTML Styles>Heading 9 style class name

-scn_htmltoc

Syntax `-scn_htmltoc`*`name`*

Parameters

`name` The name of the heading style class for the table of contents.

Description Specifies a name for the heading style class for the table of contents (the HTML element `h1`). The actual properties of this class are defined by the CSS style sheet.

Project>Options>Documentation>HTML Styles>TOC heading style class name

-sn_bt

Syntax `-sn_bt`*`name`*

Parameters

`name` The name of the body text style. By default, this is `Body Text`.

Description Determines the name of the body text style in RTF format reports. The actual properties of this style are defined by other command line options.

Project>Options>Documentation>RTF Styles>Body text style name

-sn_fpf

Syntax `-sn_fpf`*`name`*

Parameters

`name`	The name of the front page footer style. By default, this is `Front Page Footer`.

Description Determines the name of the front page footer style in RTF format reports. The actual properties of this style are defined by other command line options.

Project>Options>Documentation>RTF Styles>Front page footer style name

-sn_fph

Syntax `-sn_fph`*`name`*

Parameters

`name`	The name of the front page header style. By default, this is `Front Page Header`.

Description Determines the name of the front page header style in RTF format reports. The actual properties of this style are defined by other command line options.

Project>Options>Documentation>RTF Styles>Front page header style name

-sn_fpt

Syntax `-sn_fpt`*`name`*

Parameters

`name`	The name of the front page middle text style. By default, this is `Front Page Text`.

Description Determines the name of the main text style of the front page in RTF format reports. The actual properties of this style are defined by other command line options.

Project>Options>Documentation>RTF Styles>Front page text style name

-sn_ftr

Syntax `-sn_ftr`*name*

Parameters

name	The name of the footer style. By default, this is `Footer`.

Description Determines the name of the footer style in RTF format reports. The actual properties of this style are defined by other command line options.

Project>Options>Documentation>RTF Styles>Footer style name

-sn_hdr

Syntax `-sn_hdr`*name*

Parameters

name	The name of the header style. By default, this is `Header`.

Description Determines the name of the header style in RTF format reports. The actual properties of this style are defined by other command line options.

Project>Options>Documentation>RTF Styles>Header style name

-sn_lb1

Syntax `-sn_lb1`*name*

Parameters

name	The name of the style for top-level list bullets. By default, this is `List Bullet`.

Description Determines the name of the style for top-level list bullets in RTF format reports. The actual properties of this style are defined by other command line options.

Project>Options>Documentation>RTF Styles>List bullet 1 style name

-sn_lb2

Syntax `-sn_lb2`*`name`*

Parameters

name	The name of the style for level 2 list bullets. By default, this is `List Bullet 2`.

Description Determines the name of the style for level 2 list bullets in RTF format reports. The actual properties of this style are defined by other command line options.

Project>Options>Documentation>RTF Styles>List bullet 2 style name

-sn_lb3

Syntax `-sn_lb3`*`name`*

Parameters

name	The name of the style for level 3 list bullets. By default, this is `List Bullet 3`.

Description Determines the name of the style for level 3 list bullets in RTF format reports. The actual properties of this style are defined by other command line options.

Project>Options>Documentation>RTF Styles>List bullet 3 style name

-sn_lb4

Syntax `-sn_lb4`*`name`*

Parameters

name	The name of the style for level 4 list bullets. By default, this is `List Bullet 4`.

Description Determines the name of the style for level 4 list bullets in RTF format reports. The actual properties of this style are defined by other command line options.

Project>Options>Documentation>RTF Styles>List bullet 4 style name

-sn_lb5

Syntax `-sn_lb5`*`name`*

Parameters

name	The name of the style for level 5 list bullets. By default, this is `List Bullet 5`.

Description

Determines the name of the style for level 5 list bullets in RTF format reports. The actual properties of this style are defined by other command line options.

Project>Options>Documentation>RTF Styles>List bullet 5 style name

-sn_lb6

Syntax `-sn_lb6`*`name`*

Parameters

name	The name of the style for level 6 list bullets. By default, this is `List Bullet 6`.

Description

Determines the name of the style for level 6 list bullets in RTF format reports. The actual properties of this style are defined by other command line options.

Project>Options>Documentation>RTF Styles>List bullet 6 style name

-sn_lb7

Syntax `-sn_lb7`*`name`*

Parameters

name	The name of the style for level 7 list bullets. By default, this is `List Bullet 7`.

Description

Determines the name of the style for level 7 list bullets in RTF format reports. The actual properties of this style are defined by other command line options.

Project>Options>Documentation>RTF Styles>List bullet 7 style name

-sn_lb8

Syntax `-sn_lb8`*`name`*

Parameters

name	The name of the style for level 8 list bullets. By default, this is `List Bullet 8`.

Description

Determines the name of the style for level 8 list bullets in RTF format reports. The actual properties of this style are defined by other command line options.

Project>Options>Documentation>RTF Styles>List bullet 8 style name

-sn_lb9

Syntax `-sn_lb9`*`name`*

Parameters

name	The name of the style for level 9 list bullets. By default, this is `List Bullet 9`.

Description

Determines the name of the style for level 9 list bullets in RTF format reports. The actual properties of this style are defined by other command line options.

Project>Options>Documentation>RTF Styles>List bullet 9 style name

-sn_rtfcode

Syntax `-sn_rtfcode`*`name`*

Parameters

name	The name of the code style. By default, this is `Code`.

Description

Determines the name of the code style in RTF format reports. The actual properties of this style are defined by other command line options.

Project>Options>Documentation>RTF Styles>Code style name

-sn_rtfh1

Syntax `-sn_rtfh1`*`name`*

Parameters

name	The name of the style for top-level headings. By default, this is `Heading 1`.

Description Determines the name of the style for top-level headings in RTF format reports. The actual properties of this style are defined by other command line options.

Project>Options>Documentation>RTF Styles>Heading 1 style name

-sn_rtfh2

Syntax `-sn_rtfh2`*`name`*

Parameters

name	The name of the style for level 2 headings. By default, this is `Heading 2`.

Description Determines the name of the style for level 2 headings in RTF format reports. The actual properties of this style are defined by other command line options.

Project>Options>Documentation>RTF Styles>Heading 2 style name

-sn_rtfh3

Syntax `-sn_rtfh3`*`name`*

Parameters

name	The name of the style for level 3 headings. By default, this is `Heading 3`.

Description Determines the name of the style for level 3 headings in RTF format reports. The actual properties of this style are defined by other command line options.

Project>Options>Documentation>RTF Styles>Heading 3 style name

-sn_rtfh4

Syntax `-sn_rtfh4`*`name`*

Parameters

`name`	The name of the style for level 4 headings. By default, this is `Heading 4`.

Description Determines the name of the style for level 4 headings in RTF format reports. The actual properties of this style are defined by other command line options.

Project>Options>Documentation>RTF Styles>Heading 4 style name

-sn_rtfh5

Syntax `-sn_rtfh5`*`name`*

Parameters

`name`	The name of the style for level 5 headings. By default, this is `Heading 5`.

Description Determines the name of the style for level 5 headings in RTF format reports. The actual properties of this style are defined by other command line options.

Project>Options>Documentation>RTF Styles>Heading 5 style name

-sn_rtfh6

Syntax `-sn_rtfh6`*`name`*

Parameters

`name`	The name of the style for level 6 headings. By default, this is `Heading 6`.

Description Determines the name of the style for level 6 headings in RTF format reports. The actual properties of this style are defined by other command line options.

Project>Options>Documentation>RTF Styles>Heading 6 style name

-sn_rtfh7

Syntax `-sn_rtfh7`*`name`*

Parameters

name	The name of the style for level 7 headings. By default, this is `Heading 7`.

Description

Determines the name of the style for level 7 headings in RTF format reports. The actual properties of this style are defined by other command line options.

Project>Options>Documentation>RTF Styles>Heading 7 style name

-sn_rtfh8

Syntax `-sn_rtfh8`*`name`*

Parameters

name	The name of the style for level 8 headings. By default, this is `Heading 8`.

Description

Determines the name of the style for level 8 headings in RTF format reports. The actual properties of this style are defined by other command line options.

Project>Options>Documentation>RTF Styles>Heading 8 style name

-sn_rtfh9

Syntax `-sn_rtfh9`*`name`*

Parameters

name	The name of the style for level 9 headings. By default, this is `Heading 9`.

Description

Determines the name of the style for level 9 headings in RTF format reports. The actual properties of this style are defined by other command line options.

Project>Options>Documentation>RTF Styles>Heading 9 style name

-sn_rtftoc

Syntax `-sn_rtftoc`*`name`*

Parameters

`name`	The name of the heading style of the table of contents. By default, this is `TOC Heading`.

Description Determines the name of the heading style of the table of contents of RTF format reports. The actual properties of this style are defined by other command line options.

Project>Options>Documentation>RTF Styles>TOC heading style name

-split

Syntax `-split{0|1}`

Parameters

`0` (default)	Prints transition texts on a single line in the report.
`1`	Divides transition texts into multiple lines in the report.

Description Determines whether transition texts are divided into multiple lines in the report.

Project>Options>Documentation>Format>Split transition texts on multiple lines

-stylesheet

Syntax `-stylesheet`*`path`*

Parameters

`path`	The path to the style sheet used by HTML reports.

Description Specifies the CSS style sheet used by HTML reports.

Project>Options>Documentation>HTML Styles>Style sheet

-template

Syntax `-templatepath`

Parameters

path	The path to the style template used by RTF reports.

Description Specifies the style template used by RTF reports.

Project>Options>Documentation>RTF Styles>Style template

-test

Syntax `-test{0|1}`

Parameters

`0`	Does not include information from your testing in the report.
`1` (default)	Includes information from your testing in the report.

Description Determines whether to include information from your testing. This section contains test files such as Validator static analysis files, Validator dynamic analysis files, Validator test sequence files, and Verificator report files.

Project>Options>Documentation>Configuration>Include model test

-text_fname

Syntax `-text_fnamefont`

Parameters

font	The name of the font used for all other text than headings and code. This must exactly match the name of one of your installed fonts. By default, the value is `Times New Roman`.

Description Determines the font used for used for all other text than headings and code.

Project>Options>Documentation>Fonts>Text font name

-text_fsize

Syntax `-text_fsize`*`size`*

Parameters

`size`	An integer that represents the size in points of the font used for all other text than headings and code. By default, the value is `10`.

Description Determines the font size used for all other text than headings and code.

Project>Options>Documentation>Fonts>Text font size

-text_fstyle

Syntax `-text_fstyle{0|1|2|3}`

Parameters

`0` (default)	The text font weight is Normal.
`1`	The text font weight is Bold.
`2`	The text font weight is Italic.
`3`	The text font weight is Bold Italic.

Description Determines the weight of the font used for all other text than headings and code.

Project>Options>Documentation>Fonts>Text font style

-title

Syntax `-title`*`string`*

Parameters

`string`	The title of the report.

Description Specifies the title of the report. By default, the title of the report is the same as the name of the project.

Project>Options>Documentation>Configuration>Title

-top_margin

Syntax `-top_marginsize{cm|mm|twips|points}`

Parameters

size A decimal value that represents the size of the margin in the given unit. By default, the value is `2.5cm`.

Description Sets the top margin for the report file.

Project>Options>Documentation>Page Layout>Top margin

-toptext_justification

Syntax `-toptext_justification{0|1|2}`

Parameters

`0`	The topmost text of the front page is aligned to the left.
`1` (default)	The topmost text of the front page is centered.
`2`	The topmost text of the front page is aligned to the right.

Description Determines the alignment of the topmost text of the front page of a report in RTF format.

Project>Options>Documentation>Front Page>Top text justification

-toptext_str

Syntax `-toptext_strtext`

Parameters

text The topmost text of the front page.

Description Determines the topmost text of the front page of a report in RTF format.

Project>Options>Documentation>Front Page>Top text

-usertxtfiles

Syntax `-usertxtfiles`*`path`*`[;`*`path`*`;`*`path`*`...]`

Parameters

`path`	The path to a `txt` file to include in the report.

Description Specifies which user text files to include in the introduction section of the report.

Project>Options>Documentation>File Input>User text files

-variant

Syntax `-variant`*`name`*

Parameters

`name`	The name of the variant.

Description Specifies which variant to create a report for. By default, the Documenter creates a report for the complete model.

See also *Using variants and features*, page 217.

Use the **Variant** toolbar.

-vdafiles

Syntax `-vda`*`path`*`[;`*`path`*`;`*`path`*`...]`

Parameters

`path`	The path to a `vda` file to include in the report.

Description Specifies which Validator dynamic analysis files to include in the report.

Project>Options>Documentation>File Input>Validator dynamic analysis files

-vlgfiles

Syntax `-vlgfilespath[;path;path...]`

Parameters

path The path to a `vlg` file to include in the report.

Description Specifies which Validator test sequence files to include in the report.

Project>Options>Documentation>File Input>Validator test sequence files

-vrefiles

Syntax `-vrefilespath[;path;path...]`

Parameters

path The path to a `vre` file to include in the report.

Description Specifies which Verificator result files to include in the report.

Project>Options>Documentation>File Input>Verificator result files

-vsafiles

Syntax `-vsapath[;path;path...]`

Parameters

path The path to a `vsa` file to include in the report.

Description Specifies which Validator static analysis files to include in the report.

Project>Options>Documentation>File Input>Validator static analysis files

Part 9. Additional features and utilities

This part of the *IAR Visual State User Guide* includes these chapters:

- Prototyping a graphical interface
- Viewing design models via the Visual State Viewer
- Using IAR Visual State remotely via the Control Center
- Importing and exporting design models via XMI® files
- The Visual State State Machine API for programmatic manipulation of models
- Handling Visual State files from previous versions

Prototyping a graphical interface

- Introduction to prototyping a graphical interface
- Prototyping with Altia Design
- Graphical environment for Altia Design

Introduction to prototyping a graphical interface

Learn more about:

- *Briefly about prototyping a graphical interface*, page 883
- *Briefly about prototyping with Altia Design*, page 884
- *Briefly about prototyping based on Coder-generated code*, page 887

BRIEFLY ABOUT PROTOTYPING A GRAPHICAL INTERFACE

Many state machine models that you design and generate code for using IAR Visual State also have a graphical user interface.

If you integrate the state machine model with a model of your graphical 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 using IAR Visual State, you have two options for creating a graphical model of the user interface and integrate this model with your state machine model:

- Using the built-in support in the Validator for connecting to Altia Design (a tool for designing graphical user interfaces) and setting up the connection between your state machine model and the Altia model. This method does not require any additional programming. See *Briefly about prototyping with Altia Design*, page 884.
- Creating the graphical user interface by integrating Visual State Coder-generated code with code developed in a third-party development tool. This approach allows you to use the code for the graphical model directly in your final application. See *Prototyping based on Coder-generated code*, page 899.

BRIEFLY ABOUT PROTOTYPING WITH ALTIA DESIGN

By means of Altia Design, you can create a graphical model for your state machine model. Via the Validator you can connect the state machine model to the Altia model and simulate it.

Altia connection

An *Altia connection* is a communication link between the Validator and a graphical model created with Altia Design—an *Altia model*.

When the **Altia>Connect model** command in Validator is activated, the Validator establishes a connection to an Altia model that is automatically loaded in a new instance of Altia Design. See *Connecting a state machine model to an Altia model*, page 888.

Connections between Visual State elements and Altia objects

To use the Altia model as a user interface for the state machine model loaded in the Validator, Visual State events and action functions must be connected to Altia objects.

If you want a push button in the Altia model to generate a Visual State 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 Visual State action function turn on a LED object in the Altia model if you connect the action function and the LED object.

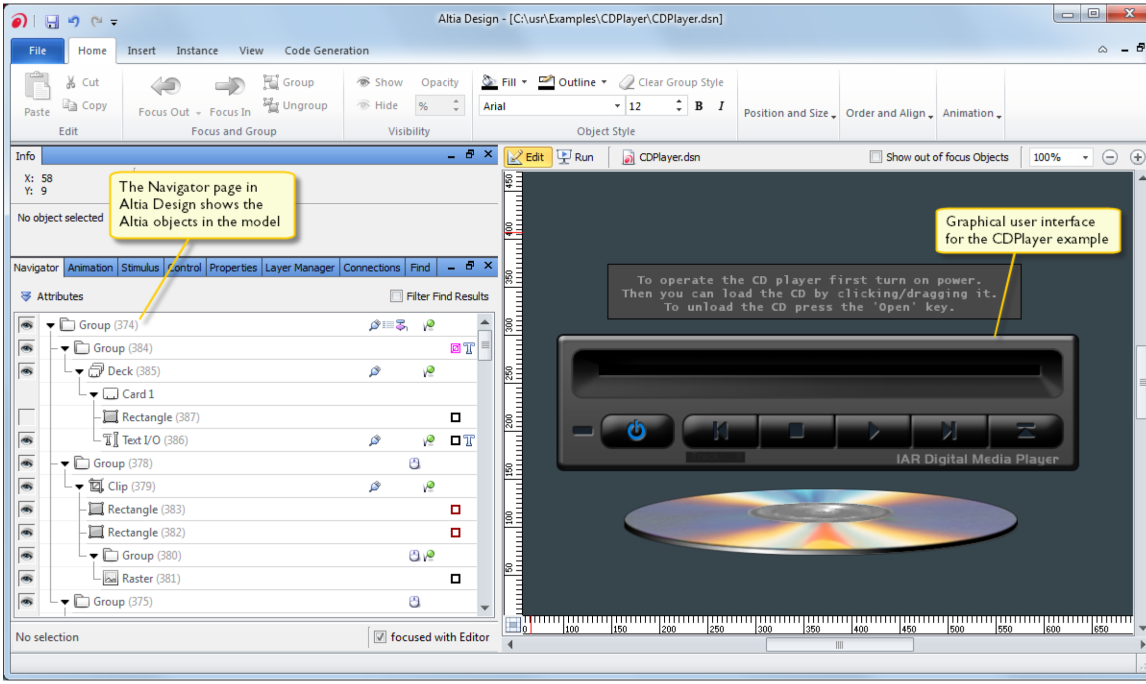
For an example of how to connect the Visual State elements to Altia objects, see *Example: Connecting Visual State elements to Altia objects for the CDplayer project*, page 893.

The objects in Altia Design are either input or output:

- Input is sent from the Altia model to the Validator, in other words, they act as events and are often bound to button objects.
- Output is sent to the Altia model as actions, for example `TurnOnLed2`.

The connections you set up are saved in the same file folder as the Altia model file (filename extension `dsn`), with the same filename but with the filename extension `vsatcons`.

This screenshot shows the Altia Design main window with the Navigator tab active, where the page shows the objects in the Altia model for the CDPlayer example:



The **Navigator** page in Altia Design shows the Altia objects in the model, and to the right, the user interface for the `CDPlayer` model is shown.

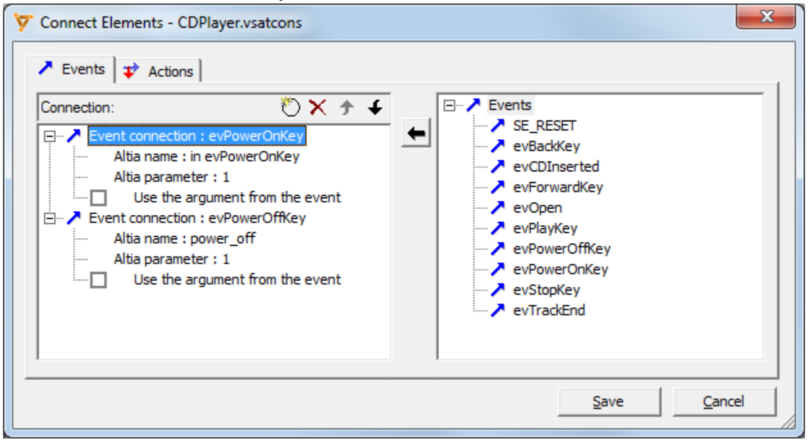
For information about how to use Altia Design, see the documentation provided with it.

Parameters on Altia objects

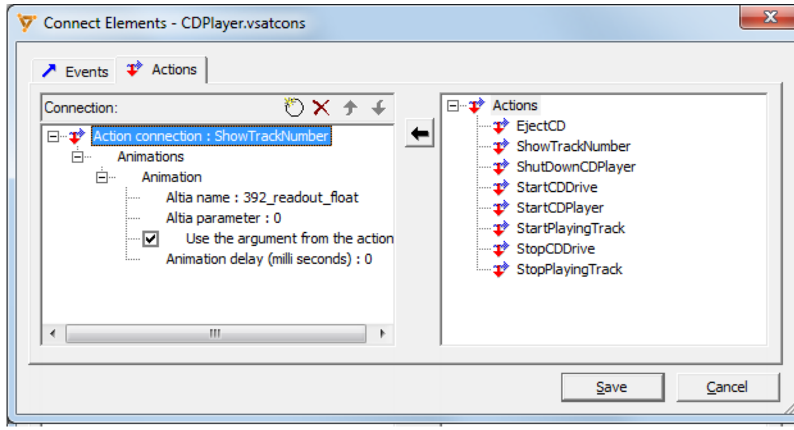
In state machine models, events and action functions are declared to carry zero or more parameters. However, Altia signals always carry one parameter, and many Altia objects accept or emit one parameter. You must consider this when you set up connections to Altia objects in the Validator.

To describe this, the example application—`CDPlayer`—provided with the IAR Visual State product installation is used. In `CDPlayer`, the `EvPowerOnKey` event has no parameter, while `EvCDInserted` is declared with one parameter. LED objects are input objects that require one parameter for which the values `0` and `1` typically mean *turn off* versus *turn on* (all parameter values for Altia design objects can be configured). Hence, to turn on a LED object, you would typically send an output signal with the parameter value `1`.

`evPowerOnKey` and `evPowerOffKey` in the `CDPlayer` example are typically connected to a graphical button. In this case a toggle button, which by default uses one as `ON` value, and zero as `OFF` value. When the button is clicked in the Altia model, a signal is sent to the Validator with the parameter 1 or 0. The combination of the name of the button in the Altia model with the parameter value 1 must then be connected to `evPowerOnKey`, and a connection with the name of the button with the name and the parameter value 0 must be connected to the event `evPowerOff`. In some cases, some other name than the name of the button can be used as the item to assign a value to in the Altia model. This can be the case if some variable is used in the Altia model and you want to use that value directly:



Some action functions in the Validator might send arguments as well. If you want to, you can use the argument sent from the Validator and pass that on to the Altia model, or you can specify that some other argument should be sent to Altia. This is an example of sending the argument from the Validator to the Altia model:

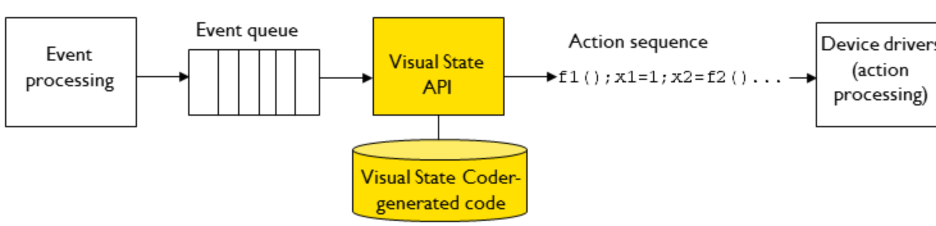


BRIEFLY ABOUT PROTOTYPING BASED ON CODER-GENERATED CODE

You can create a software graphical model of your Visual State model using the Coder-generated code directly in any third-party development tool that supports Standard C/C++/C#/Java code.

The control logic code is generated by the Coder. By means of the Visual State APIs, it can be combined with code developed with any third-party development tool that supports the programming language used when generating the code by the Coder.

You implement the prototype as you would implement a final application. This means that you can reuse the control logic designed in Visual State 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 this figure:



Creating a prototype in Microsoft Visual C++ differs from creating one in a console application in how the Visual State event deduction sequence is implemented.

Implementing an infinite while 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 Visual State events.
- Using separate threads.

For an example, see *Prototyping based on Coder-generated code*, page 899.

Prototyping with Altia Design

What do you want to do?

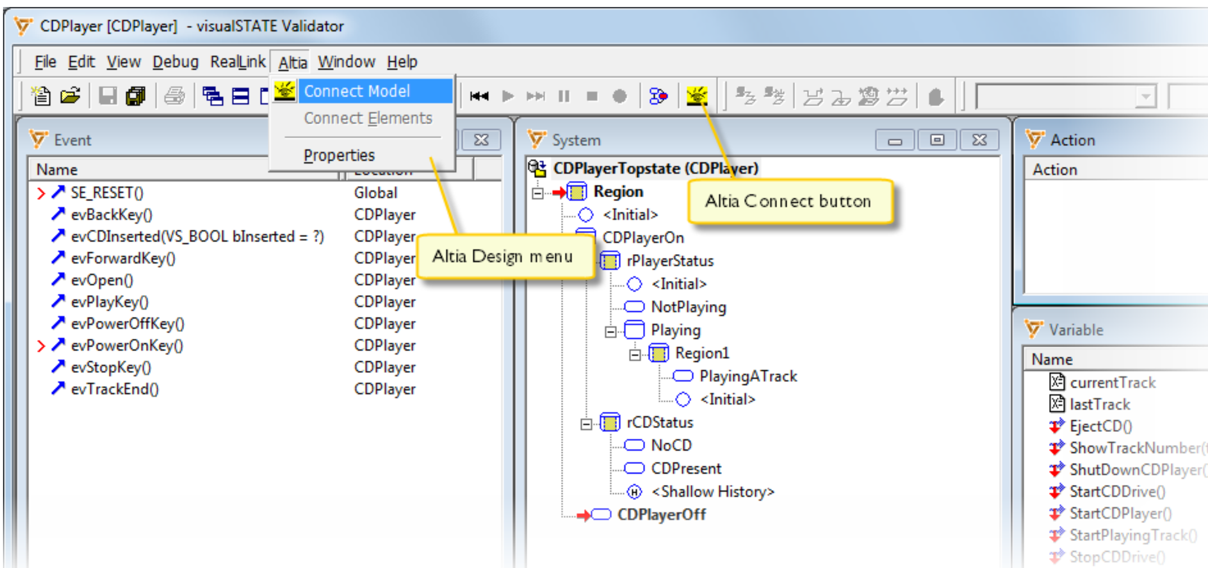
- *Connecting a state machine model to an Altia model*, page 888
- *Connecting Visual State elements to Altia objects*, page 890
- *Removing a connection between an Altia object and a Visual State element*, page 892
- *Simulating with Altia Design*, page 892
- *Closing the Altia connection*, page 892
- *Configuring the Altia connection*, page 893
- *Example: Connecting Visual State elements to Altia objects for the CDplayer project*, page 893
- *Prototyping based on Coder-generated code*, page 899

CONNECTING A STATE MACHINE MODEL TO AN ALTIA MODEL

To simulate your Visual State design model using an Altia model, you must first establish a connection between the two via the Validator.

1 Start the Validator and load the state machine model that you want to simulate.

2 Choose **Altia>Connect Model** or click the Altia **Connect** button.



3 In the **Open Altia Design** dialog box, select the Altia model to connect to. Choose between:

- If the desired Altia model is listed in **Open Most Recently used Altia Design**, select it from the list
- 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 model in a new instance of Altia Design.
- Or create a new design—click **Create a New Altia Design**, and then click **OK** to open an empty Altia editor. Here you can create the new Altia model right away while the Altia connection is active. For information on how to use the Altia editor, see the documentation provided with Altia Design.

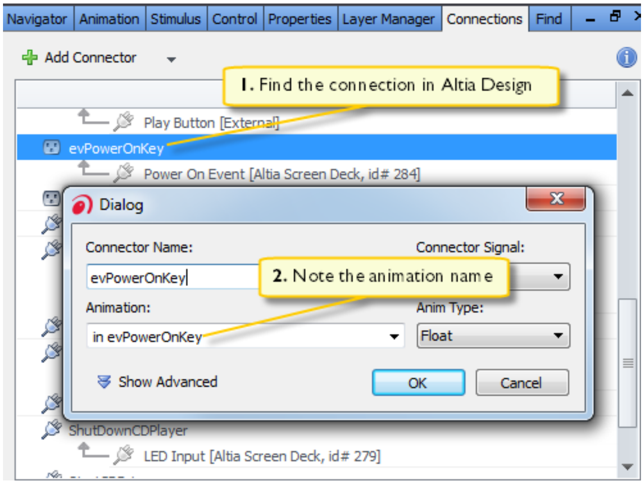
Whether you connect to an existing Altia model or create a new one, it is possible to edit it while the Altia connection is active. Any design changes will have immediate effect in the Validator, for example adding new objects and connecting them to the state machine model through new or existing external signal connection. See *Connecting Visual State elements to Altia objects*, page 890.

You might even choose to create only the parts of the Altia model that you want to simulate at the moment and maybe add more objects later.

CONNECTING VISUAL STATE ELEMENTS TO ALTIA OBJECTS

You can use this procedure for connecting both events and actions to Altia objects.

1 In Altia Design, find the name of the Altia object that you want to connect to a Visual State event. In this example, `evPowerOnKey`:



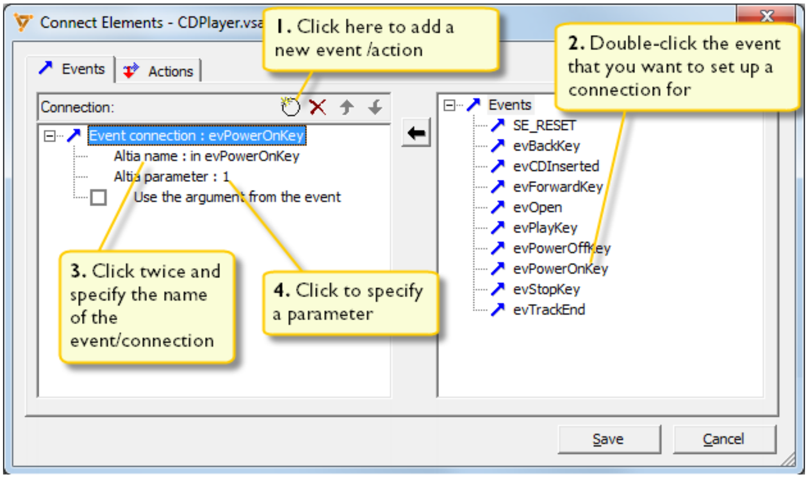
Note: If you already know the name of the Altia object you do not need to perform this step.

2 In the Validator, choose **Altia>Connect Elements**, and click the **Events** tab or the **Actions** tab to set up a connection with an event or an action, respectively.

3 Click the **New** button to add a new event/action connection and perform these steps:

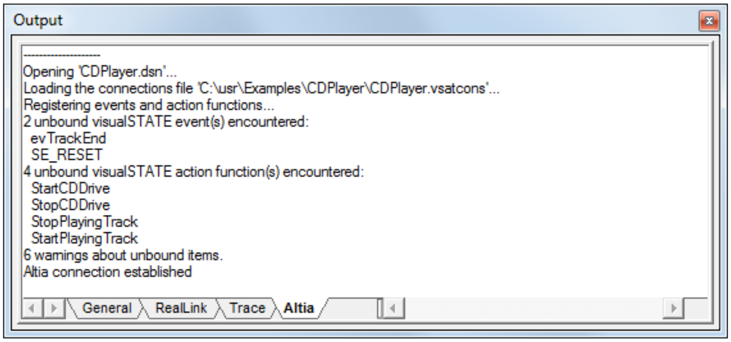
- Double-click the event, or action, in the list to the right that you want to set up a connection for. In this example, `evPowerOnKey`.
- In the **Connection** pane, click the Altia event/action row twice (or press F2) and specify the name of the event/action, which is also the name of the connection in the Altia model.
- If any parameters are needed, click twice (or press F2) on the Altia parameter row and enter the argument to use. For a power on button, this could typically be `1` for off.

It should now look, for example like this:



4 When you have connected your Visual State element to an Altia object, the names of the Visual State events and action functions will be added to the Altia model as new external signals if they are not already there.

The events and action functions that are not connected to any object in the Altia model are listed in the Validator **Output** window as `unbound Visual State events` and `unbound Visual State action functions`.



The reporting of unbound events and action functions are done when you connect to the Altia model, and when you click **Save** in the **Connect Elements** dialog box.

See also *Example: Connecting Visual State elements to Altia objects for the CDplayer project*, page 893

REMOVING A CONNECTION BETWEEN AN ALTIA OBJECT AND A VISUAL STATE ELEMENT

1 Open your state machine model in the Validator.

2 Choose **Altia>Connect Model**.

3 Choose **Altia>Connect Elements**.

4 Select the event or action connection you want to remove and click the **Delete** button to delete the connection. Click **Save**. The editing of the connections will have effect on the connection at once.

SIMULATING WITH ALTIA DESIGN

When you have connected your state machine model to an Altia model, you can start simulation. You can start the simulation even if you have not created a complete Altia model.

1 In Altia Design, choose **Set Run Mode** from the menu or press Ctrl+D.

2 To simulate events, you can use these two methods:

- In the Validator, double-click the event name in the **Event** window

  When you send an event to the Visual State system using the Validator, the event is also sent to the Altia model where the connected input object is *animated* accordingly, provided that the object type supports animation. For example toggle buttons will change from `OFF` to `ON`.

- In the Altia model, manipulate the corresponding object.

Note: When Altia Design is in edit mode, you cannot manipulate event generators such as buttons in the Altia model, and thus no events will be sent from Altia to the design model in the Validator.

3 Action functions that are executed in the Validator and connected to an Altia object will have a visible effect in the Altia model, for example turning on a LED.

Note: Action functions executed in guard expressions and assignments will have no visible effect in the Altia model.

CLOSING THE ALTIA CONNECTION

1 When you are finished using the Altia model, click the **Connect/Disconnect to/from Altia** toolbar button (), or choose **Altia>Disconnect** in the Validator to close the Altia connection.

The Altia connection will also be closed automatically when the Validator is closed.

2 Closing the Altia connection does not close Altia Design. When you open an Altia connection again, a new Altia Design instance is created.

CONFIGURING THE ALTIA CONNECTION

Typically, the default values of the Altia connection works as is. However, the Altia connection can be configured to suit specific needs.

1 Choose **Altia>Properties** to open the **Define Altia Properties** dialog box.

2 Make your settings in the dialog box. For reference information, see *Define Altia Properties dialog box*, page 905.

Note: To ensure synchronization between the state machine model and the Altia model, select the options **Reset Altia design when deducting SE_RESET** and **Always initialize and reset the state machine model**.

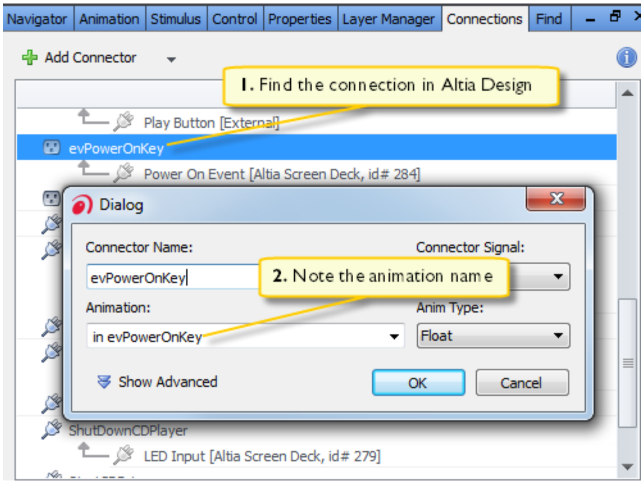
3 When you are finished, click **OK**.

EXAMPLE: CONNECTING VISUAL STATE ELEMENTS TO ALTIA OBJECTS FOR THE CDPLAYER PROJECT

This example procedure uses the `CDPlayer` project as a base for describing how to connect some Visual State elements to Altia objects.

1 In the Validator, choose **Altia>Connect Model** to set up a connection between your state machine model and your Altia model.

2 In Altia Design, find the name of the Altia object that you want to connect to a Visual State event. In this example, `evPowerOnKey`:



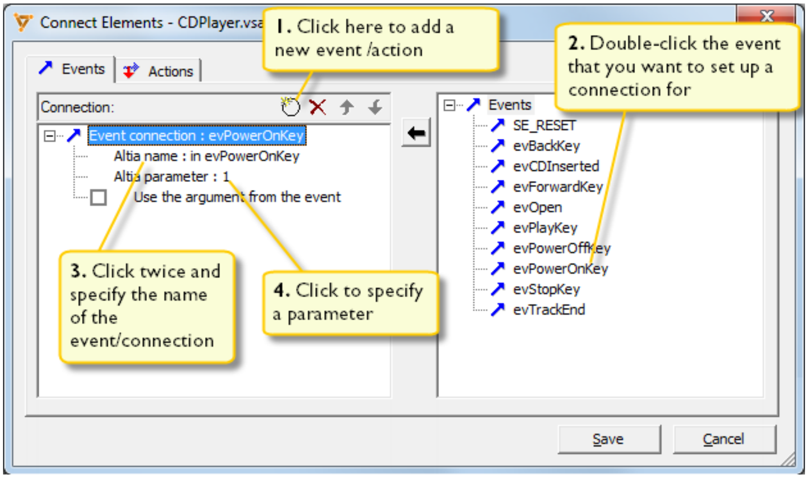
Note: If you already know the name of the Altia object you do not need to perform this step.

3 In the Validator, choose **Altia>Connect Elements**, and click the **Events** tab.

4 Click the **New** button to add a new event connection and perform these steps:

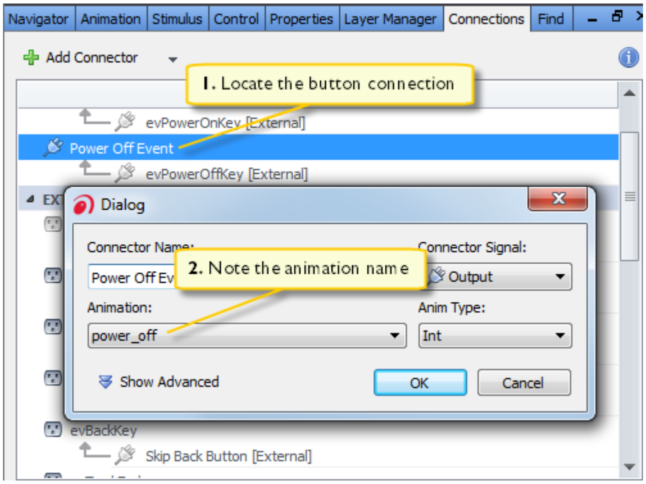
- Double-click `evPowerOnKey` in the list of events to the right.
- Click the Altia event name twice and specify `evPowerOnKey` which is also the name of the connection in the Altia model.
- Because this button uses 1 as On, click twice on the Altia parameter row and enter 1 as the argument to use.

It should now look like this:

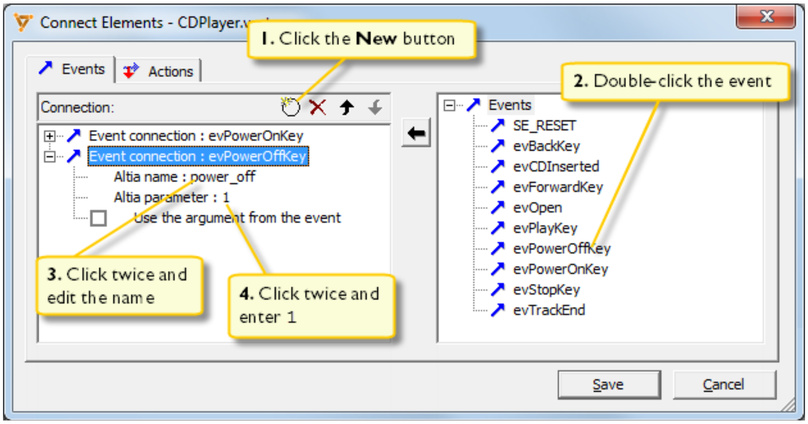


5 If you want to, you can also make an event connection directly to a variable value change in the Altia model. If you make such an event connection, the Validator will respond by taking the Visual State event when it is signaled from the Altia model that the variable is set to the value specified in the event connection. And vice versa - when you double-click the event that is on the event connection inside the Validator, the Altia model will be signaled from the Validator, that the variable should now be set to the value indicated in the event connection.

For CDPlayer, there is an external connection for the power off button, as well as an animation called power_off:



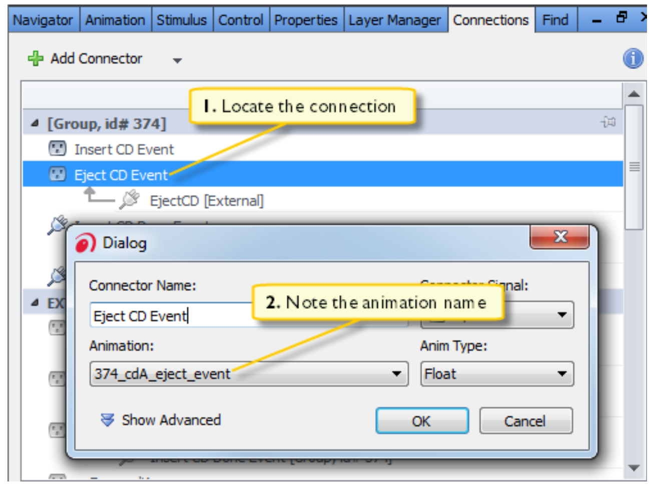
6 To make an event connection that matches this, choose **Altia>Connect Elements** and click the **Events** tab. Create a new event connection. Select the event evPowerOffKey, specify the Altia event name to power_off, and set the Altia parameter to 1.



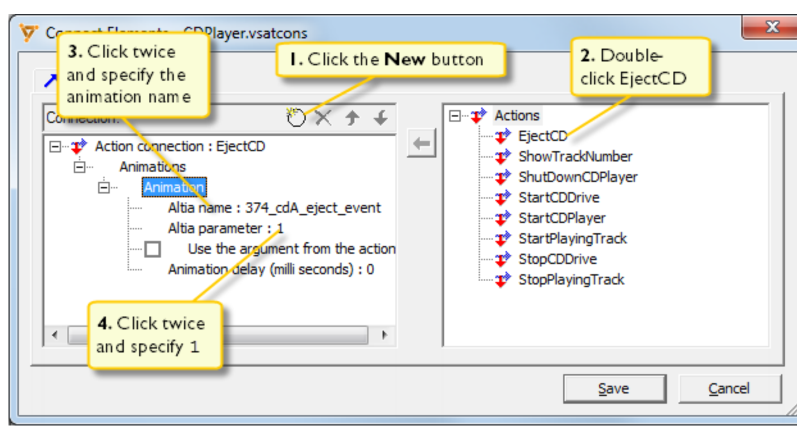
Click **Save**.

7 If you want an animation of your Altia model when an action function is called in the Validator, you should set up an action connection.

For the `CDPlayer` example, the CD could be ejected when the Validator calls the action function `EjectCD`. In the Altia model, the animation can be seen below:



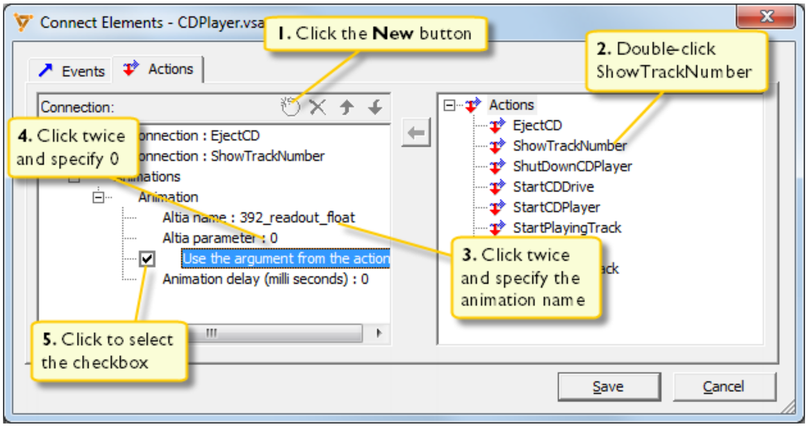
8 To set up an action connection to do this, choose **Altia>Connect Elements** and click the **Actions** tab. Double click `EjectCD` in the list to the right. Then specify the Altia name to `374_cdA_eject_event` and also edit the Altia parameter to be `1`.



Click **Save**.

9 In some cases you want to have the state machine model signal to the Altia model what to display. For `CDPlayer`, you might want to get the track number from the Validator model.

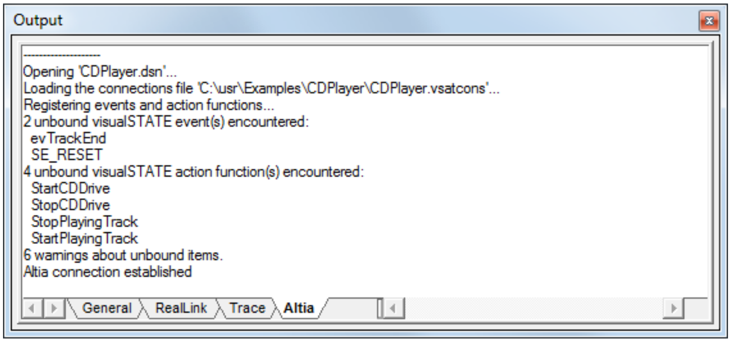
To set up an action connection in the Validator to do this, choose **Altia>Edit Connections** click the **Actions** tab. Double-click the `ShowTrackNumber` to the right. Edit the Altia name to be `392_readout_float`, edit the Altia parameter to be `0`, and select **Use the argument from the action**.



Click **Save**.

10 When you have connected your state machine model to an Altia model, the names of the Visual State events and actions functions will be added to it as new external signals if they are not already there.

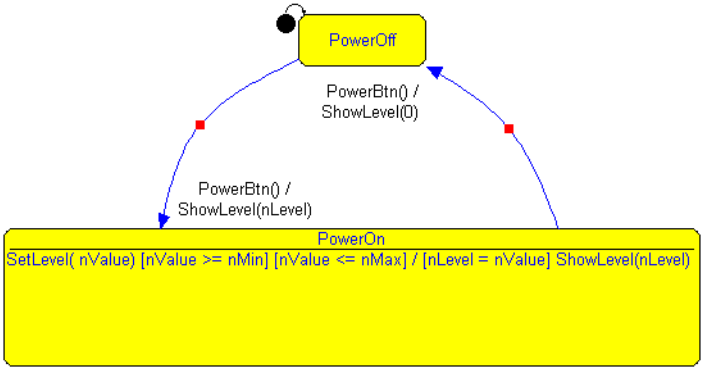
The events and action functions that are not connected to any object in the graphical model are listed in the Validator **Output** window as `unbound Visual State events` and `unbound Visual State action functions`.



The reporting of unbound events and action functions are done when you connect to the Altia model, and when you click **Save** in the **Edit Connections** window.

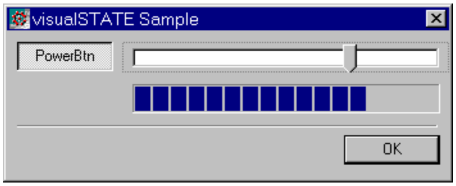
PROTOTYPING BASED ON CODER-GENERATED CODE

This is an example of how a graphical model can be implemented by capturing the Windows idle message. The example is based on a state machine with two states: `PowerOn` and `PowerOff`:



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 for displaying 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:



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.

To implement this in C++ code:

1. Include the Coder-generated code files in your Visual C++ project. Remember to disable the **Precompiled Headers** option for these files, because you are including C files in a C++ project.
2. Define an event queue for adding and retrieving events. For an example, see the example code included with Visual State.

3 Initialize the controls with the constants defined in IAR Visual State and initialize the Visual State system in the `OnInitDialog` function like this:

```
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_InitAll();

     // 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** button's click command to the function `OnPowerBtn`. Map the slider control's 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 to 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 to the
     // queue
     QueueElement qe;
     qe.event     = SetLevel;
     qe.parameter = m_hSlider.GetPos();
     add(qe);
```

5 Define the implementation of the Visual State action `ShowLevel` like this:

```
VS_VOID ShowLevel(VS_INT nValue)
{
     // get a handle to the main dialog box
     CMainDlg* pDlg = (CMainDlg*)AfxGetMainWnd();
     ASSERT(pDlg);
     // force the dialog box to update the progress bar
     pDlg->SetProgressPos(nValue);
}
```

6 Implement the Visual State 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 VSDeduct with the event
        unsigned char cc;
        switch(hQe.event) {
             case SE_RESET :
                   cc = VSDeduct(SE_RESET);
                   break;
             case PowerBtn :
                   cc = VSDeduct(hQe.event);
                   break;
             case SetLevel :
                   cc = VSDeduct(hQe.event, hQe.parameter);
                   break;
             default       :
                   cc = -1; // unknown event
                   break;
        }
```

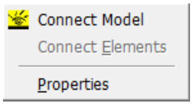
```
        If ((cc != SES_OKAY) && (cc != SES_FOUND))
             ; // Error handler
      }
      return 0L;
}
```

Graphical environment for Altia Design

Reference information about:

Altia menu

The **Altia** menu provides commands for connecting to Altia Design:



Menu commands

These commands are available on the menu:

Connect Model

Displays the **Open Altia Model** dialog box where you can choose an existing Altia model or create a new model, and then connect to Altia Design. See *Open Altia Model dialog box*, page 906.

Connect Elements

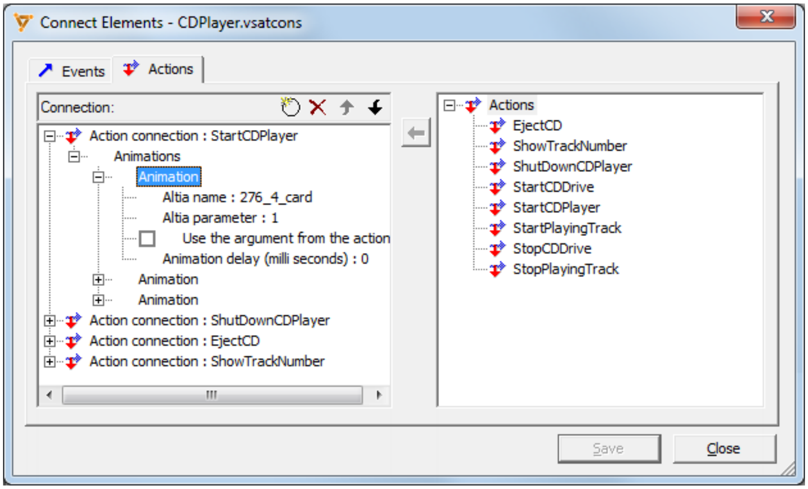
Displays a dialog box, see *Connect Elements dialog box*, page 903.

Properties

Displays the **Define Altia Properties** dialog box, see *Define Altia Properties dialog box*, page 905.

Connect Elements dialog box

The **Connect Elements** dialog box is available from the **Altia** menu in the Validator.



Use this dialog box to set up connections between Altia objects and Visual State events and action functions.

See also *Connecting Visual State elements to Altia objects*, page 890.

Connection (for events)

Lists all connections between Visual State events and Altia objects with these details:

Event connection	The name of the event in the Validator.
Altia event	Specify the name of the object in the Altia model that the Visual State event should be connected to.
Altia parameter	Specify the value to be sent from the Altia model for the given Altia event, or that will be sent to the Altia model, when the event in the Validator is sent to the state machine model.
Use the argument from the event	Makes the Validator send the first argument from the event in the Validator to the Altia model, when the event is activated in the Validator. If not selected, the value from the Altia parameter will be sent.

Click a value to edit it (or use F2).

Connection (for actions)

Lists all connections between Visual State action functions and Altia objects with these details:

Action connection	The name of the action function in the Validator.
Altia name	Specify the name of the object in the Altia model that the Visual State action function should be connected to.
Altia parameter	Specify the value to be sent from the Altia model for the given Altia action function, when the action function is called in the Validator.
Use the argument from the action	Makes the Validator send the first argument from the action function in the Validator to the Altia model, when the action function is called in the Validator. If not selected, the value from the Altia parameter will be sent.
Animation delay	The delay after the animation has been sent to the Altia model. In other words, the Validator will make a pause for the given time before sending more animation values to the Altia model.

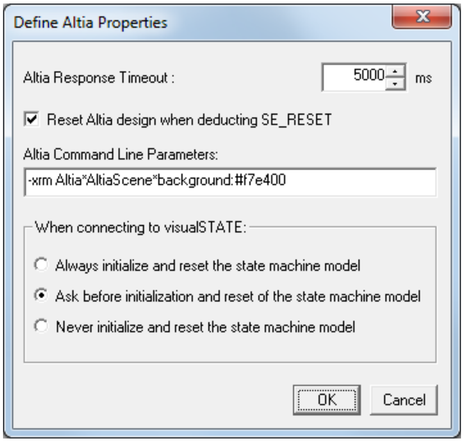
Click a value to edit it (or use F2).

Pane to the right

A list of all events or action functions of your state machine model, for which you can make a connection to your Altia model. Select the event or action function you want to connect to an object, and click the arrow button. The event/action function appears in the **Connection** pane.

Define Altia Properties dialog box

The **Define Altia Properties** dialog box is available from the **Altia** menu.



Use this dialog box to configure the Altia connection between a state machine model and an Altia model.

Altia Response Timeout

Specify the number of milliseconds that the Validator waits for a response from Altia Design before timing out.

Reset Altia design when deducting SE_RESET

Synchronizes the Altia model with the state machine model when the Visual State reset event `SE_RESET` is deducted.

Altia Command Line Parameters

Type any arguments to pass to Altia Design. For a description of recognized parameters, see the documentation provided with Altia Design.

When connecting to Visual State

Specify whether the state machine model should be initialized and reset when connecting to Altia. Choose between:

Always initialize and reset the state machine model

Resets and initializes the state machine model automatically when connecting to the Altia model.

Ask before initialization and reset of the state machine model

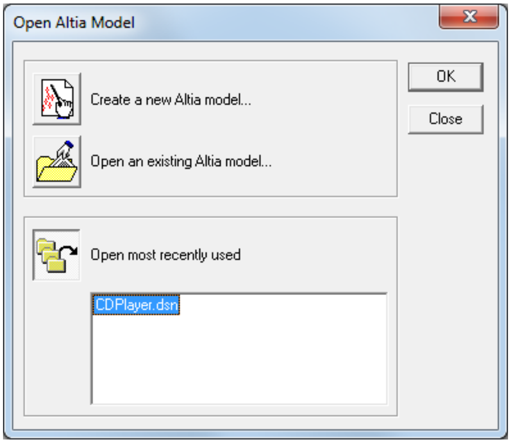
Prompts you when connecting to the Altia model to let you decide whether to reset and initialize the state machine model.

Never initialize and reset the state machine model

Connects to the Altia model without resetting or initializing the state machine model.

Open Altia Model dialog box

The **Open Altia Model** dialog box is available by choosing **Altia>Connect Model** in the Validator.



Use this dialog box to choose an existing Altia model or create a new model, and then connect to Altia Design.

See also *Connecting a state machine model to an Altia model*, page 888.

Create a new Altia model

Opens a dialog box where you can create a new Altia model.

Open an existing Altia model

Opens a standard file browser dialog box where you can locate an existing Altia model and open it.

Open most recently used

Opens the most recently used Altia model.

Viewing design models via the Visual State Viewer

Introduction to the Visual State Viewer

Learn more about:

- *Briefly about the Visual State Viewer*, page 907

BRIEFLY ABOUT THE VISUAL STATE VIEWER

The Visual State Viewer is a stand-alone application that can be used for viewing all state machine models made by using the Designer without having access to the Visual State product. This is useful for sharing and showing design ideas to someone who does not need to edit the models, for example sales staff or third-party companies. The Viewer can show and print your state machine diagrams.

The Viewer does not require a license and does not depend on any other Visual State files. It only requires some common runtime DLLs from the operating system, so you can move a copy of the `Viewer.exe` file wherever you want.

The `Viewer.exe` file can be found in the `bin` directory in your IAR Visual State product installation.

If you want someone to view your state machine model, you should give them a copy of your model files and a copy of `Viewer.exe`.

For example, for the `AVSystem` example that is provided with IAR Visual State, you should give a copy of the these files:

```
AVSystem.vsp
CDPlayer.vsr
Viewer.exe
```

Using IAR Visual State remotely via the Control Center

- Introduction to the Visual State Control Center
- Using the Control Center

Introduction to the Visual State Control Center

Learn more about:

- *Briefly about the Visual State Control Center*, page 909

BRIEFLY ABOUT THE VISUAL STATE CONTROL CENTER

The Control Center is a stand-alone application that can handle a set of commands and take appropriate actions as opening an application remotely, or forwarding a request to the Designer or the Validator. For the Designer you can quickly create new projects and with some initial states. For the Validator you can send events to the state machine model being simulated.

In addition, you can use the Control Center for invoking external tools, for example tools for creating advanced graphical user interfaces, like the CGI Studio Scene Composer.

The Control Center must be started before any command can be sent to it, and it manages command requests and responses by means of the JSON–RPC format. TCP/IP is used for the communication. Normally, the Control Center listens on port 8090 and it puts no restrictions on who sends commands to it.

You can find the `ControlCenter.exe` file in the `bin` directory in the IAR Visual State product installation.

Using the Control Center

What do you want to do?

- *Starting the Control Center*, page 910
- *Saving all files in connected applications*, page 910

STARTING THE CONTROL CENTER

1. Open a command prompt.
2. Start the Control Center with the path to the IAR Visual State product installation, `ControlCenter.exe`
3. Optionally, you can:
   - use `ControlCenter.exe -port`*`nnnn`* to communicate using port *`nnnn`*.
   - use `ControlCenter.exe -verbose` to display status information on the screen.

SAVING ALL FILES IN CONNECTED APPLICATIONS

When you have a number of applications connected to the Control Center, you can save the files in the connected applications remotely.

1. Send a command to the Control Center. Use this as an example of how to save all files:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "saveAll",
}
```

2 The Control Center replies with a message that indicates success or error. In case of success, the reply might look like this:

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": true
}
```

EXITING THE CONTROL CENTER

1 To exit the Control Center, and optionally also all connected applications, send a command to the Control Center with these arguments:

`all`	A Boolean value that determines whether all connected applications should exit or not.

2 Use this as an example of how to exit the Control Center and all connected applications:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "quit",
  "params":
  {"all": "true"}
}
```

3 The Control Center replies with a message that indicates success or error. In case of success, the reply might look like this:

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": true
}
```

STARTING THE DESIGNER

You can start the Designer remotely, loading an existing project, and optionally set focus to a specific item.

1 To start the Designer remotely, you can send a command to the Control Center with these arguments:

`projectPath`	The full path to the project to load.

`focusItemGuid`	Optional guid for the item to set focus on. If not used, the Designer will choose what to focus on.

2 Use this as an example of how to start the Designer:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "launchDesigner",
  "params": {
    "projectPath": "d:/Test/JSONTestProject/JSONTestProject.vsp",
    "focusItemGuid": "1985B3C4-74A3-42B5-B03E-941B13633A5C"
  }
}
```

3 The Designer replies through the Control Center with a message that indicates success or error. In case of success, the reply might look like this:

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": true
}
```

SAVING IN THE DESIGNER

When you have started the Designer, you can make sure that you save open files in it.

1 To save the files the Designer has loaded, send a command to the Control Center with these arguments:

`projectPath`	The full path to the loaded project.

2 Use this as an example of how to save in the Designer:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "saveDesigner",
  "params":
  {"projectPath": "d:/Test/JSONTestProject/JSONTestProject.vsp"}
}
```

3 The Designer replies through the Control Center with a message that indicates success or error. In case of success, the reply might look like this:

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": true
}
```

EXITING THE DESIGNER

You can exit the Designer remotely, without receiving a prompt to save open files.

1 To exit the Designer, send a command to the Control Center with these arguments:

`projectPath`	The full path to the loaded project.

2 Use this as an example of how to exit the Designer:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "quitDesigner",
  "params":
  {"projectPath": "d:/Test/JSONTestProject/JSONTestProject.vsp"}
}
```

3 The Designer replies through the Control Center with a message that indicates success or error. In case of success, the reply might look like this:

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": true
}
```

CREATING A PROJECT WITH A NEW STATE MACHINE

The Designer can be remotely controlled to create a new project with a new system, a new top-level state machine, and a state machine below it.

1 To start the Designer remotely, you can send a command to the Control Center with these arguments:

`projectPath`	The full path to the project to create.
`projectName`	The name of the project. The name must be a legal identifier in C.

`projectGuid`	Optional guid (Global Unique Identifier) to use for the new project. If not used, a new guid is automatically assigned to the project.
`systemName`	The name of the new system. The name must be a legal identifier in C.
`systemGuid`	Optional guid to use for the new system. If not used, a new guid is automatically assigned to the system.
`topStatePath`	The full path to the top-level state machine file to create.
`topStateName`	The name of the new top-level state machine. The name must be a legal identifier in C.
`topStateGuid`	Optional guid to use for the new top-level state machine. If not used, a new guid is automatically assigned to the top-level state machine.
`stateMachineName`	The name of the new state machine. The name must be a legal identifier in C.
`stateMachineGuid`	Optional guid to use for the new state machine. If not used, a new guid is automatically assigned to the state machine.

2 Use this as an example for how to start the Designer and create a new state machine:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "createProjectAndStateMachine",
  "params": {
    "projectPath": "d:/Test/TestJSON/TestJSONProject.vsp",
    "projectName": "TestProject",
    "projectGuid": "1234-5678-9012",
    "systemName": "System0",
    "systemGuid": "1234-5678-9013",
    "topStatePath": "d:/Test/TestJSON/TestJSONProject.vsr",
    "topStateName": "Topstate1",
    "topStateGuid": "1234-5678-9014",
    "stateMachineName": "State1",
    "stateMachineGuid": "1234-5678-9015"
  }
}
```

3 The Designer replies through the Control Center with a message that indicates success or error. In case of success, the reply might look like this:

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": true
}
```

ADDING A STATE MACHINE TO AN EXISTING DESIGN

The Designer can be started remotely to load an existing project, and add a new state to a parent region or state.

1 To start the Designer remotely, you can send a command to the Control Center with these arguments:

`projectPath`	The full path to the project to load and modify.
`parentGuid`	Guid to use for the parent to add a new state to.
`newStateMachineName`	Optional name of the new state to add. If not specified, the Designer will choose a new name.
`newStateMachineGuid`	Optional guid to use for the new item to add. If not used, a new guid is automatically assigned to the new state.

2 Use this as an example for how to add a new state to a specific parent:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "addStateMachine",
  "params": {
    "projectPath": "d:/Test/TestJSON/TestJSONProject.vsp",
    "parentGuid": "1234-5678-9015",
    "newStateMachineName": "NewState",
    "newStateMachineGuid": "1234-5678-9016"
  }
}
```

3 The Designer replies through the Control Center with a message that indicates success or error. In case of success, the reply might look like this:

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": true
}
```

STARTING THE VALIDATOR

The Validator can be started remotely to load an existing project and optionally set focus to a specific item.

1 To start the Validator remotely, you can send a command to the Control Center with these arguments:

`projectPath`	The full path to the project to load.
`focusItemGuid`	Optional guid for the item to set focus on. If not used, the Validator will choose what to focus on.

2 Use this as an example for how to start the Validator:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "launchValidator",
  "params": {
    "projectPath": "d:/Test/JSONTestProject/JSONTestProject.vsp",
    "focusItemGuid": "1985B3C4-74A3-42B5-B03E-941B13633A5C"
  }
}
```

3 The Validator replies through the Control Center with a message that indicates success or error. In case of success, the reply might look like this:

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": true
}
```

SAVING IN THE VALIDATOR

When you have started the Validator, you can make sure that you save open files in it.

1 To save the files the Validator has loaded, send a command to the Control Center with these arguments:

`projectPath`	The full path to the loaded project.

2 Use this as an example of how to save in the Validator:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "saveValidator",
  "params":
  {"projectPath": "d:/Test/JSONTestProject/JSONTestProject.vsp"}
}
```

3 The Validator replies through the Control Center with a message that indicates success or error. In case of success, the reply might look like this:

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": true
}
```

EXITING THE VALIDATOR

You can exit the Validator remotely, without receiving a prompt to save open files.

1 To exit the Validator, send a command to the Control Center with these arguments:

`projectPath`	The full path to the loaded project.

2 Use this as an example of how to exit the Validator:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "quitValidator",
  "params":
  {"projectPath": "d:/Test/JSONTestProject/JSONTestProject.vsp"}
}
```

3 The Validator replies through the Control Center with a message that indicates success or error. In case of success, the reply might look like this:

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": true
}
```

DISABLING LOOK AHEAD OF GUARD VALUES IN THE VALIDATOR

When the Validator is simulating, it can be set up to show the values of guard expressions. That means the Validator will evaluate all guard expressions after each deduction step. However, in the case of remote simulation, this might be undesirable. The Validator can be set up to disable this look ahead for guard expressions during the session being remotely controlled.

1 To change the Validator setup to enable or disable the look ahead for guard expressions, you can send a command to the Control Center with these arguments:

`projectPath`	The full path to the project to load.
`disable`	True or false to indicate whether to disable or not.

2 Use this as an example for disabling look ahead of guard values in the Validator:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "disableLookAheadGuardCheck",
  "params": {
    "projectPath": "d:/Test/JSONTestProject/JSONTestProject.vsp",
    "disable": true
  }
}
```

3 The Validator replies through the Control Center with a message that indicates success or error. In case of success, the reply might look like this:

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": true
}
```

DEDUCING AN EVENT IN THE VALIDATOR

The Validator can be started remotely to perform a simulation step with a specific event and optionally arguments for the event. The Validator will perform the simulation step and reply with any action function calls that are to be performed and their optional arguments. In the meantime, the Validator can send a request for getting action function return values to determine whether guard expression is true or false, of what value to assign to some variable.

1 To make the Validator perform a simulation step, you can send a command to the Control Center with these arguments:

`projectPath`	The full path to the project to load.
`eventName`	The name of the event to deduce for the given model.
`eventArguments`	Optional argument. If used, it must be an array of values.

2 Use this as an example for how to call the Validator and make it perform a simulation step for an event:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "deductEvent",
  "params": {
    "projectPath": "d:/Test/JSONTestProject/JSONTestProject.vsp",
    "eventName": "SE_RESET"
  }
}
```

3 The Validator replies through the Control Center with a message that indicates success or error, and can in this case reply with this answer which indicates that the action function call `Action0(2,0)` is to be performed:

```
{
  "id": 2,
  "jsonrpc": "2.0",
  "result": {
  "actions": [
    {
      "action": {
        "actionArguments": [2,0],
        "actionName": "Action0"
        }
      }
    ]
  }
}
```

REQUESTING AN ACTION FUNCTION CALL RETURN VALUE FROM THE VALIDATOR

The Validator might, as part of performing a simulation step with the `deductEvent` command, ask for the return value from some action function call. In that case, the Validator will send a request back to the caller that initiated the `deductEvent` command to get the value from the action function call.

1 When the Validator needs the return value for an action function call, it will send a command with these arguments:

`actionName`	The name of the action function to get the return value for.
`actionArguments`	Optional array with arguments for the action function call asked for.

2 Use this as an example of a sequence with the `deductEvent` command causing the Validator to send a `getActionFunctionCallResult` command back.

The initial command sent from the client:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "deductEvent",
  "params": {
    "projectPath": "d:/Test/JSONTestProject/JSONTestProject.vsp",
    "eventName": "Event4"
  }
}
```

3 The command sent from the Validator to get the return value:

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "method": "getActionFunctionCallResult",
  "params": {
    "action": {
      "actionArguments": [2,0],
      "actionName": "Action3"
    }
  }
}
```

4 This is the reply from the client with the return value to use:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": 7
}
```

5 This is the result of the complete simulation step sent from the Validator to the client:

```
{
  "id": 2,
  "jsonrpc": "2.0",
  "result": {
    "actions": [
      {
        "assignment": {
          "assignedTo": "External1",
          "assignedToIndex": null,
          "value": 7
        }
      },
      {
        "action": {
          "actionArguments": null,
          "actionName": "Action5"
        }
      },
      {
        "assignment": {
          "assignedTo": "ExternalArray",
          "assignedToIndex": 2,
          "value": 7
        }
      }
    ]
  }
}
```

SIMULATING A VALIDATOR PROJECT REMOTELY

You can use the Control Center to simulate a project remotely in the Validator. The steps below give an example of how to simulate a project called `AVSystem`, located in the directory `e:\AVSystem`.

1 Start the Control Center, located (for example) in this directory:

```
c:\Program Files (x86)\IAR Systems\Visual State 8.n\bin\ControlCe
nter.exe
```

If the Control Center needs access through the firewall, you will be prompted to allow it. Then, a window is opened displaying the text `Visual State Control Center version x.x.x.xxxx`.

2 You might need to enable `telnet`, which is disabled by default. Open a command prompt with administrator privileges and type `telnet /?`. If it returns a description for `telnet`, you can continue. Then run:

```
telnet localhost 8090
```

If the connection is successful, the Control Center window displays `connected to 127.0.0.1`.

3 For most commands sent to the Control Center, a reply like this indicates success:

```
{"id":1,"jsonrpc":"2.0","result":true}
```

4 Start the Validator using this command:

```
{"jsonrpc": "2.0","id": 1,"method": "launchValidator","params":
{"projectPath": "e:/AVSystem/AVSystem.vsp"}
}
```

Note that new line characters might cause problems with `telnet`.

5 Disable the guard lookahead:

```
{"jsonrpc": "2.0", "id": 1, "method":
"disableLookAheadGuardCheck", "params":
{"projectPath": "e:/AVSystem/AVSystem.vsp", "disable": true}
}
```

6 Send the event `SE_RESET`:

```
{"jsonrpc": "2.0", "id": 2, "method": "deductEvent", "params":
{"projectPath": "e:/AVSystem/AVSystem.vsp", "eventName":
"SE_RESET"}
}
```

The reply indicates which actions that were called. In this case, no actions:

```
{"id":2,"jsonrpc":"2.0","result":{"actions":null}}
```

7 Send the event `evPowerKey`:

```
{"jsonrpc": "2.0", "id": 2, "method": "deductEvent", "params":
{"projectPath": "e:/AVSystem/AVSystem.vsp", "eventName":
"evPowerKey"}
}
```

The reply is:

```
{"id":2,"jsonrpc":"2.0","result":{"actions":[{"action":{"actionAr
guments":null,"actionName":"StartCdPlayer"}}]}}
```

8 Send the event `evDetect` to indicate that there is a CD-player:

```
{"jsonrpc": "2.0", "id": 2, "method": "deductEvent", "params":
{"projectPath": "e:/AVSystem/AVSystem.vsp", "eventName":
"evDetect", "eventArguments": [1]}
}
```

The reply is:

```
{"id":2,"jsonrpc":"2.0","result":{"actions":null}}
```

9 Start playing:

```
{"jsonrpc": "2.0", "id": 2, "method": "deductEvent", "params":
{"projectPath": "e:/AVSystem/AVSystem.vsp", "eventName":
"evPlayKey"}
}
```

This prompts for the return value for the call to `FindLastTrack()`:

```
{"id":1,"jsonrpc":"2.0","method":"getActionFunctionCallResult","p
arams":{"action":
{"actionArguments":null,"actionName":"FindLastTrack"}
}}
```

As a reply to that, you can send:

```
{"jsonrpc": "2.0", "id": 1, "result": 3}
```

and get the list:

```
{"id":2,"jsonrpc":"2.0","result":{"actions":[{"assignment":{"assi
gnedTo":"lastTrack","assignedToIndex":null,"value":3}},{"assignme
nt":{"assignedTo":"currentTrack","assignedToIndex":null,"value":0
}},{"action":{"actionArguments":null,"actionName":"LocateTrackSta
rt"}}]}}
```

10 Send an event to tell that you found the track start:

```
{"jsonrpc": "2.0", "id": 2, "method": "deductEvent", "params":
{"projectPath": "e:/AVSystem/AVSystem.vsp", "eventName":
"evFoundTrackStart"}
}
```

The reply is:

```
{"id":2,"jsonrpc":"2.0","result":{"actions":[{"action":{"actionAr
guments":null,"actionName":"StartPlayingTrack"}}]}}
```

11 Send an event to tell that the track end was reached:

```
{"jsonrpc": "2.0", "id": 2, "method": "deductEvent", "params":
{"projectPath": "e:/AVSystem/AVSystem.vsp", "eventName":
"evTrackEnd"}
}
```

The reply is:

```
{"id":2,"jsonrpc":"2.0","result":{"actions":[{"action":{"actionAr
guments":null,"actionName":"StopPlayingTrack"}},{"assignment":{"a
ssignedTo":"currentTrack","assignedToIndex":null,"value":1}},{"ac
tion":{"actionArguments":null,"actionName":"LocateTrackStart"}}]}
}
```

12 Continue to send events as appropriate.

13 Exit the Validator:

```
{"jsonrpc": "2.0","id": 1,"method": "quitValidator","params":
{"projectPath": "e:/AVSystem/AVSystem.vsp"}
}
```

The reply is:

```
{"id":1,"jsonrpc":"2.0","result":true}
```

For information about other ways to exit a connected application, see the full list of commands.

STARTING EXTERNAL TOOLS VIA THE CONTROL CENTER

You can use the Control Center for invoking external tools. This example shows how to invoke the CGI Studio Scene Composer.

1 To start the Scene Composer remotely, you can send a command to the Control Center with these arguments:

`projectPath`	The full path to the project to load.

`guidToFocusGuid`	Guid of a state to set focus on in the Scene Composer.

2 Use this as an example for how to start the CGI Studio Scene Composer remotely:

```
{
  "jsonrpc": "2.0",
  "method": "launchSceneComposer",
  "params": {
    "projectPath": "d:/Test/TestJSON/TestJSONProject.vsp",
    "guidToFocus": "1234-5678-9015"
  }
}
```

Importing and exporting design models via XMI® files

- Introduction to using the XMI file format
- Using the XMI format for import and export of design models

Introduction to using the XMI file format

Learn more about:

- *Briefly about the XMI file format*, page 927
- *Restrictions and requirements for importing XMI files to IAR Visual State*, page 927
- *Restrictions and requirements for exporting XMI files from IAR Visual State*, page 928

BRIEFLY ABOUT THE XMI FILE FORMAT

XMI (XML Metadata Interchange) is a file format specified by the Object Management Group, intended for tool-independent exchange of design models. This means that you can transfer state machine models between IAR Visual State and tools by vendors other than IAR Systems.

RESTRICTIONS AND REQUIREMENTS FOR IMPORTING XMI FILES TO IAR VISUAL STATE

The mapping from UML to Visual State state machine elements is one-to-one, but some exceptions apply. These exceptions, and some other properties of the XMI import to keep in mind, are:

- The XMI file to be imported must conform to UML 2.1.
- Any other files referred to from the XMI file are ignored; only the state machine models directly included in the XMI file are imported. This means that when exporting from another UML tool, the UML design should be exported to a single file.
- The XMI file to be imported must contain profile information, otherwise the XMI import might miss some or all state machine components.

- States that specify instance termination are mapped to (region-local) final states in IAR Visual State.
- IAR Visual State computes a graphical layout automatically. (Any graphical layout information in the XMI file to be imported is ignored.)
- Transition guards are imported as is, without parsing them or even trying to map them to any Visual State elements.
- Transition actions are imported as is and become explanations in IAR Visual State.
- Every transition that crosses one or more state boundary is rendered as a pair of transitions using connector states, where these transitions do not cross any state boundaries. (All transitions that do not cross any state boundaries are imported as they are.)

RESTRICTIONS AND REQUIREMENTS FOR EXPORTING XMI FILES FROM IAR VISUAL STATE

The mapping from IAR Visual State to UML state machine elements is one-to-one, but some exceptions apply. These exceptions, and some other properties of the XMI export to keep in mind, are:

- The generated XMI file conforms to UML 2.1.
- Models that contain submachine states are not exported, because IAR Visual State's support for arbitrary bindings has no counterpart in UML.
- Layout information is not exported. (Tools that import XMI automatically compute a layout. The strategies used in that differ from tool to tool.)
- Signals are mapped to a macro, recognized by some UML tools, for sending an event between state machines. (This might not always be the desired behavior.)
- Some UML modeling tools cannot handle regions and parallelism inside a state.
- Positive and negative state conditions are not exported, because they have no counterpart in UML.
- Explanations are attached to the corresponding UML element if allowed. If not, explanations are attached to the closest UML element, moving upwards in the model hierarchy, that can carry explanations.

Visual State types map to UML types like this:

Visual State type	UML type
VSBool	BOOL
VSInt	LONG
VSInt16	SHORT

Table 40: Mapping from Visual State types to UML types

Visual State type	UML type
VSInt32	LONG
VSUChar	UNSIGNED_CHAR
VSUInt	UNSIGNED_LONG
VSUInt16	UNSIGNED_SHORT
VSUInt32	UNSIGNED_LONG
VSVoid	VOID
VSVoidPtr	VOID_PTR
VSFloat	FLOAT
VSDouble	DOUBLE

Table 40: Mapping from Visual State types to UML types

Using the XMI format for import and export of design models

What do you want to do?

- *Importing an XMI file to IAR Visual State*, page 929
- *Exporting an XMI file from IAR Visual State*, page 929

IMPORTING AN XMI FILE TO IAR VISUAL STATE

If you have a state machine model created in a tool from another vendor, you can use it in IAR Visual State if the other tool supports the XMI format.

1 In the Designer, choose **File>Open** and filter the view by **State Machine Files in XMI Format (*.xmi)**.

2 Browse to the XMI file you want to import and click **Open**.

Only the state machine parts of a UML design are imported; no class structure, no diagram information, no explanations, etc.

EXPORTING AN XMI FILE FROM IAR VISUAL STATE

If you have a state machine model created using IAR Visual State, you can export it and then open it in a tool from another vendor if the other tool supports the XMI format.

1 In the Designer, choose **File>Open** and open the state machine model that you want to export to XMI format.

2 Make sure that the state that represents the correct top-level state machine is selected in the **Project Browser** window and choose **File>Save As**.

3 Choose **State Machine Files in XMI Format (*.xmi)** from the **Save as type** drop-down menu and save the file.

A class structure is created to hold the state machine that corresponds to the exported top-level state machine. Exported events, action functions, and variables are generated at appropriate places in the exported XMI structure.

The Visual State State Machine API for programmatic manipulation of models

Introduction to the State Machine API and programmatic manipulation

Learn more about:

- *Briefly about the Visual State State Machine API*, page 931
- *Installed files*, page 932

BRIEFLY ABOUT THE VISUAL STATE STATE MACHINE API

With the Visual State State Machine API (which has a C interface) you can manipulate and extract all parts of your state machine design model.

By calling the API, you can programmatically access and change your state machine models from various programming languages that support calling C functions based on some foreign function binding mechanism.

For example, you can use the API to:

- Add new items to a project, or create a new project
- Delete items from a project
- Rename items in a project
- Extract copies of parts from a project to build a representation of the items for your own purpose
- Search for items in a project.

From the API you can also add *tags* to items in the project. A tag is a pair of strings—a name and a value. They look, for example, like this: `Requirement Reference` and `Section 5.4.6`. Tags can only be manipulated by the API, but the Visual State components keep the tags persistent.

When you work with the API you do not need to set exact positions for new items that you add. When the Designer loads a model with any item that does not have a set position, it will position it for you, and you can move it to the point you like. The API preserves the positions for items that have a set position.

The State Machine API DLL is stand-alone, in other words, it does not depend on any other Visual State DLLs, or any runtime DLLs except the usual runtime DLLs for the operating system.

Using the API does not require a license, so you can freely copy and use it as you like.

INSTALLED FILES

The API is delivered as a set of header files, a dynamic link library (DLL), and a set of generated documentation files in HTML format. The DLL is built with "C" linkage, so it can be accessed from most programming languages.

The files for the API and the generated documentation can be found in the `doc\StateMachineAPI` directory in the IAR Visual State product installation. The generated documentation contains a number of examples of use cases, examples of how to access the API from C and C++. The main documentation file is `doc\StateMachineAPI\html\index.html`.

Handling Visual State files from previous versions

- Introduction to using old design models from previous versions

Introduction to using old design models from previous versions

Learn more about:

- *Using files from version 5 and later*, page 933
- *Converting old files by using the Navigator*, page 933
- *Converting old files manually by using the project converter*, page 933

USING FILES FROM VERSION 5 AND LATER

The file format used in version 5 and 6 of IAR Visual State is not the same format as being used by version 7.4, and later. When you load an old project, the Navigator can convert the files, or you can convert the files manually by running a program from a command prompt.

CONVERTING OLD FILES BY USING THE NAVIGATOR

1. Before you start, make sure to have backup copies of your files.
2. In the Navigator, open the workspace for the old project that you want to convert.
3. The Navigator will ask you if you want to convert and save the project. Answer Yes.
4. The Navigator saves the old files in a `backup` directory, converts them, and saves the converted files in the same directory that you loaded your workspace from.

   You can now use your newly converted files in the new version of IAR Visual State.

CONVERTING OLD FILES MANUALLY BY USING THE PROJECT CONVERTER

1. Before you start, make sure to have backup copies of your files.
2. Open a command prompt.
3. Change the directory to where you have your old project.
4. To start the conversion, use this command line:

   `ProjectConverter.exe` *`project`*`.vsp` *`converted`*

Where:

project.vsp	is the name of your old project.
converted	is the name of the destination folder for your converted project.

5 `ProjectConverter` converts the old files and places the converted files in the directory you specified.

You can now use your newly converted files in the new version of IAR Visual State.

Glossary

This is a glossary for terms relevant to embedded systems programming in general, and to IAR Visual State® and state machine design specifically.

A

Application
The program developed by the user of the IAR Systems toolkit and which will be run as an embedded application on a target processor.

Architecture
A term used by computer designers to designate the structure of complex information-processing systems. It includes the kinds of instructions and data used, the memory organization and addressing, and the methods by which the system is implemented. The two main architecture types used in processor design are *Harvard architecture* and *von Neumann architecture*.

Argument
Arguments are the values provided for the (formal) parameters when invoking a function, template, or macro, etc. Arguments are also referred to as *actual parameters*. Compare *Parameter*.

Auto variables
The term refers to the fact that each time the function in which the variable is declared is called, a new instance of the variable is created automatically. This can be compared with the behavior of local variables in systems using static overlay, where a local variable only exists in one instance, even if the function is called recursively. Also called local variables. Compare *Register variables*.

B

Batch files
A text file containing operating system commands which are executed by the command line interpreter. In Unix, this is called a "shell script" because it is the Unix shell which includes the command line interpreter. Batch files can be used as a simple way to combine existing commands into new commands.

Bitfield
A group of bits considered as a unit.

Breakpoint
A breakpoint in IAR Visual State is a specification of one or more conditions that will cause the `Deduct` function to break, and wait for acknowledge before continuing. A breakpoint might contain conditions that, if true, will cause a break before a transition is taken (a pre-condition) and conditions that, if true, will cause a break after a transition has been taken (a post condition). Breakpoints can be used in C-SPYLink and in the Validator.

C

Calling convention
A calling convention describes the way one function in a program calls another function. This includes how register parameters are handled, how the return value is returned, and which registers that will be preserved by the called function. The compiler handles this automatically for all C and C++ functions. All code written in assembler language must conform to the rules in the calling convention to be callable from C or C++, or to be able to call C and C++ functions. The C calling convention and the C++ calling conventions are not necessarily the same.

Code pointers
A code pointer is a function pointer. Compilers often provide several different code pointers to support microcontrollers that allow several different methods of calling a function. Compilers for embedded systems usually provide the users with the ability to use all these methods.

Do not confuse code pointers with data pointers.

Compilation unit
See *Translation unit*.

Compiler options
Parameters you can specify to change the default behavior of the compiler.

Composite state
A state that consists of concurrent regions, or mutually exclusive states.

Context menu
A context menu appears when you right-click in the user interface, and provides context-specific menu commands.

C-style preprocessor
A preprocessor is either a stand-alone application or an integrated part of a compiler, that performs preprocessing of the input stream before the actual compilation occurs. A C-style preprocessor follows the rules set up in Standard C and implements commands like `#define`, `#if`, and `#include`, which are used to handle textual macro substitution, conditional compilation, and inclusion of other files.

D

Data pointers
Many cores have different addressing modes to access different memory types or address spaces. Compilers for embedded systems usually have a set of different data pointer types so they can access the available memory efficiently.

Data representation
How different data types are laid out in memory and what value ranges they represent.

Declaration
A specification to the compiler that an object, a variable or function, exists. The object itself must be defined in exactly one translation unit (source file). An object must be declared before the object can be referred to. Normally an object that is used in many files is defined in one source file. A declaration is normally placed in a header file that is included by the files that use the object.

For example:

```
/* Variable "a" exists somewhere. Function
 "b" takes two int parameters and returns an
 int. */

extern int a;
int b(int, int);
```

Definition
The variable or function itself. Only one definition can exist for each variable or function in an application. See also *Tentative definition*.

For example:

```
int a;
int b(int x, int y)
{
   return x + y;
}
```

Device driver
A piece of software that acts as an interface to hardware devices, to make it possible for application software to use the hardware device without detailed knowledge of the exact design of the device.

Dynamic initialization
Variables in a program written in C are initialized during the initial phase of execution, before the main function is called. These variables are always initialized with a static value, which is determined either at compile time or at link time. This is called static initialization. In C++, variables might require initialization to be performed by executing code, for example, running the constructor of global objects, or performing dynamic memory allocation. The latter is called dynamic initialization.

Dynamic memory allocation
There are two main strategies for storing variables: statically at link time, or dynamically at runtime. Dynamic memory allocation is often performed from the heap and it is the size of

the heap that determines how much memory that can be used for dynamic objects and variables. The advantage of dynamic memory allocation is that several variables or objects that are not active at the same time can be stored in the same memory, thus reducing the memory requirements of an application. See also *Heap memory*.

Dynamic object
An object that is allocated, created, destroyed, and released at runtime. Dynamic objects are almost always stored in memory that is dynamically allocated. Compare *Static object*.

E

Element file
See *Transition element file*.

Embedded C++
A subset of the C++ programming language, which is intended for embedded systems programming. The fact that performance and portability are particularly important in embedded systems development was considered when defining the language.

Embedded system
A combination of hardware and software, designed for a specific purpose. Embedded systems are often part of a larger system or product.

Emulator
An emulator is a hardware device that performs emulation of one or more derivatives of a processor family. An emulator can often be used instead of the actual core and connects directly to the printed circuit board—where the core would have been connected—via a connecting device. An emulator always behaves exactly as the processor it emulates, and is used when the debugging requires all systems actuators, or when debugging device drivers.

Enumeration
A type which includes in its definition an exhaustive list of possible values for variables of that type. Common examples include Boolean, which takes values from the list [true, false], and day-of-week which takes values [Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday]. Enumerated types are a feature of typed languages, including C and Ada.

Executable image
Contains the executable image; the result of linking several relocatable object files and libraries.

Exception
An exception is an interrupt initiated by the processor hardware, or hardware that is tightly coupled with the processor, for instance, a memory management unit (MMU). The exception signals a violation of the rules of the architecture (access to protected memory), or an extreme error condition (division by zero).

Do not confuse this use of the word exception with the term *exception* used in the C++ language (but not in Embedded C++).

Extended keywords
Non-standard keywords in C and C++. These usually control the definition and declaration of objects (that is, data and functions). See also *Keywords*.

F

Filling
How to fill up bytes—with a specific fill pattern—that exists between the sections in an executable image. These bytes exist because of the alignment demands on the sections.

Format specifiers
Used to specify the format of strings sent by library functions such as `printf`. In the following example, the function call contains one format string with one format specifier, `%c`, that prints the value of `a` as a single ASCII character:

```
printf("a = %c", a);
```

G

General options
Parameters that you can specify to change the default behavior of all tools that are included in IAR Visual State.

Generic pointers
Pointers that have the ability to point to all different memory types in, for example, a core based on the Harvard architecture.

Global element
An event, action, variable, signal, etc, that is defined at project level. Thus, it has the scope of the Visual State project, including all Visual State systems contained in it.

H

Harvard architecture
A core based on the Harvard architecture has separate data and instruction buses. This allows execution to occur in parallel. As an instruction is being fetched, the current instruction is executing on the data bus. Once the current instruction is complete, the next instruction is ready to go. This theoretically allows for much faster execution than a von Neumann architecture, but adds some silicon complexity. Compare *von Neumann architecture*.

Heap memory
The heap is a pool of memory that is reserved for dynamic memory allocation. An application can request parts of the heap for its own use; once memory is allocated from the heap it remains valid until it is explicitly released back to the heap by the application. This type of memory is useful when the number of objects is not known until the application executes. Note that this type of memory is risky to use in systems with a limited amount of memory or systems that are expected to run for a very long time.

Heap size
Total size of memory that can be dynamically allocated.

High-level device driver
A device driver written to control external peripheral units, such as displays, etc. Compare *Device driver* and *High-level device driver*.

Host
The computer that communicates with the target processor. The term is used to distinguish the computer on which the debugger is running from the core that the embedded application you develop runs on.

I

IDE (integrated development environment)
A programming environment with all necessary tools integrated into one single application.

Image
See *Executable image*.

Include file
A text file which is included into a source file. This is often done by the preprocessor.

Initialized sections
Read/write sections that should be initialized with specific values at startup.

Inlining
An optimization that replaces function calls with the body of the called function. This optimization increases the execution speed and can even reduce the size of the generated code.

Interrupt vector
A small piece of code that will be executed, or a pointer that points to code that will be executed when an interrupt occurs.

Interrupt vector table
A table containing interrupt vectors, indexed by interrupt type. This table contains the processor's mapping between interrupts and interrupt service routines and must be initialized by the programmer.

Interrupts
In embedded systems, the use of interrupts is a method of detecting external events immediately, for example a timer overflow or the pressing of a button.

Interrupts are asynchronous events that suspend normal processing and temporarily divert the flow of control through an "interrupt handler" routine. Interrupts can be caused by both hardware (I/O, timer, machine check) and software (supervisor, system call, or trap instruction). Compare *Trap*.

Intrinsic
An adjective describing native compiler objects, properties, events, and methods.

Intrinsic functions
1. Function calls that are directly expanded into specific sequences of machine code. 2. Functions called by the compiler for internal purposes (that is, floating-point arithmetic etc.).

K

Key bindings
Key shortcuts for menu commands used in IAR Visual State.

Keywords
A fixed set of symbols built into the syntax of a programming language. All keywords used in a language are reserved—they cannot be used as identifiers (in other words, user-defined objects such as variables or procedures). See also *Extended keywords*.

L

L-value
A value that can be found on the left side of an assignment and thus be changed. This includes plain variables and dereferenced pointers. Expressions like `(x + 10)` cannot be assigned a new value and are therefore not L-values.

Language extensions
Target-specific extensions to the C language.

Library configuration file
A file that contains a configuration of the runtime library. The file contains information about what functionality is part of the runtime environment. The file is used for tailoring a build of a runtime library.

Local element
An event, action, variable, signal, etc, that is defined at top-level state machine level. It normally has the scope of the top-level state machine itself.

Local variable
See *Auto variables*.

Low-level device driver
A device driver written to control a chip's on-board peripheral units, such as A/D, timers, etc. Compare *Device driver* and *High-level device driver*.

M

Macro
1. An assembler macro is user-defined sets of assembler lines that can be expanded later in the source file by referring to the given macro name. Parameters will be substituted if referred to.

2. A C macro is a text substitution mechanism used during preprocessing of source files. Macros are defined using the `#define` preprocessing directive. The replacement text of each macro is then substituted for any occurrences of the macro name in the rest of the translation unit.

3. A C-SPY macro is a program that you can write to enhance the functionality of C-SPY. A typical application of C-SPY macros is to associate them with breakpoints; when such a breakpoint is hit, the macro is run and can for example be used to simulate peripheral devices, to evaluate complex conditions, or to output trace data.

The C-SPY macro language is like a simple dialect of C, but is less strict with types.

Mailbox
A mailbox in an RTOS is a point of communication between two or more tasks. One task can send messages to another task by placing the message in the mailbox of the other task. Mailboxes are also known as message queues or message ports.

Memory area
A region of the memory.

Memory bank
The smallest unit of continuous memory in banked memory. One memory bank at a time is visible in a core's physical address space.

Memory map
A map of the different memory areas available to the core.

Memory model
Specifies the memory hierarchy and how much memory the system can handle. Your application must use only one memory model at a time, and the same model must be used by all user modules and all library modules.

Microcontroller
A microprocessor on a single integrated circuit intended to operate as an embedded system. In addition to a CPU, a microcontroller typically includes small amounts of RAM, PROM, timers, and I/O ports.

Microprocessor
A CPU contained on one (or a few) integrated circuits. A single-chip microprocessor can include other components such as memory, memory management, caches, floating-point unit, I/O ports and timers. Such devices are also known as microcontrollers.

Module, link
Normally, the result of compiling a single translation unit. A module consists of, for example, symbol definitions, references, code, data, and relocation information. An object file usually contains one module. See *Translation unit* and *Object file, relocatable*.

N

Navigator workspace
A logical representation for handling a collection of projects, systems, and state machine diagrams, and their files. The workspace contains session-specific information. It is stored in a file with the filename extension `vnw`.

Non-volatile storage
Memory devices such as battery-backed RAM, ROM, magnetic tape and magnetic disks that can retain data when electric power is shut off. Compare *Volatile storage*.

O

Object
An object file or a library member.

Object file, absolute
See *Executable image*.

Object file, relocatable
The result of compiling or assembling a source file. See *Module, link*.

Operator
A symbol used as a function, with infix syntax if it has two arguments (`+`, for example) or prefix syntax if it has only one (for instance, bitwise negation, ~). Many languages use operators for built-in functions such as arithmetic and logic.

Operator precedence
Each operator has a precedence number assigned to it that determines the order in which the operator and its operands are evaluated. The highest precedence operators are evaluated first. Use parentheses to group operators and operands to control the order in which the expressions are evaluated.

Options
A set of commands that control the behavior of a tool, for example the compiler or linker. The options can be specified on the command line or in IAR Visual State.

Output image
See *Executable image*.

P

Parameter
Parameters are used in the definition of a function, template, or macro. Parameters are also referred to as *formal parameters*. Compare *Argument*.

Parameter passing
See *Calling convention*.

Peripheral unit
A hardware component other than the processor, for example memory or an I/O device.

Pointer
An object that contains an address to another object of a specified type.

#pragma
During compilation of a C/C++ program, the #pragma preprocessing directive causes the compiler to behave in an implementation-defined manner. This can include, for example, producing output on the console, changing the declaration of a subsequent object, changing the optimization level, or enabling/disabling language extensions.

Preemptive multitasking
An RTOS task is allowed to run until a higher priority process is activated. The higher priority task might become active as the result of an interrupt. The term preemptive indicates that although a task is allotted to run a given length of time (a timeslice), it might lose the processor at any time. Each time an interrupt occurs, the task scheduler looks for the highest priority task that is active and switches to that task. If the located task is different from the task that was executing before the interrupt, the previous task is suspended at the point of interruption.

Compare *Round Robin.*

Preprocessing directives
A set of directives that are executed before the parsing of the actual code is started.

Preprocessor
See *C-style preprocessor*.

Project / Visual State project
A collection of systems. Each project can contain several state machine diagrams in addition to global elements. The project data is stored in a file with the filename extension `vsp`.

Project options
General options that apply to an entire project, for example the signal queue mode which is a project option. The signal queue mode can be set on a project in the Designer.

PROM
Programmable Read-Only Memory. A type of ROM that can be programmed only once.

Q

Qualifiers
See *Type qualifiers*.

R

Real-time operating system (RTOS)
An operating system which guarantees the latency between an interrupt being triggered and the interrupt handler starting, and how tasks are scheduled. An RTOS is typically much smaller than a normal desktop operating system. Compare *Real-time system*.

Real-time system
A computer system whose processes are time-sensitive. Compare *Real-time operating system (RTOS)*.

Region
A region defines concurrent subsystems and represents hierarchical state machines.

Register
A small on-chip memory unit, usually just one or a few bytes in size, which is particularly efficient to access and therefore often reserved as a temporary storage area during program execution.

Register variables
Typically, register variables are local variables that are placed in registers instead of on the (stack) frame of the function. Register variables are much more efficient than other variables because they do not require memory accesses, so the compiler can use shorter/faster instructions when working with them. See also *Auto variables*.

Reset
A reset is a restart from the initial state of a system. A reset can originate from hardware (hard reset), or from software (soft reset). A hard reset can usually not be distinguished from the power-on condition, which a soft reset can be. In IAR Visual State, reset is typically related to sending `SE_RESET` to a system, which means that the Visual State system is reset.

ROM-monitor
A piece of embedded software designed specifically for use as a debugging tool. It resides in the ROM of the evaluation board chip and communicates with a debugger via a serial port or network connection. The ROM-monitor provides a set of primitive commands to view and modify memory locations and registers, create and remove breakpoints, and execute your application. The debugger combines these primitives to fulfill higher-level requests like program download and single-step.

Round Robin
Task scheduling in an operating system, where all tasks have the same priority level and are executed in turn, one after the other. Compare *Preemptive multitasking*.

RTOS
See *Real-time operating system (RTOS)*.

R-value
A value that can be found on the right side of an assignment. This is just a plain value. See also *L-value*.

S

Saturation arithmetics
Most, if not all, C and C++ implementations use mod-2^N 2-complement-based arithmetics where an overflow wraps the value in the value domain, that is (127+1)=-128. Saturation arithmetics, on the other hand, does *not* allow wrapping in the value domain, for instance, (127+1)=127, if 127 is the upper limit. Saturation arithmetics is often used in signal processing, when an overflow condition would have been fatal if wrapping had been allowed.

Short addressing
Many cores have special addressing modes for efficient access to internal RAM and memory-mapped I/O. Short addressing is therefore provided as an extended feature by many compilers for embedded systems. See also *Data pointers*.

Side effect
An expression in C or C++ is said to have a side-effect if it changes the state of the system. Examples are assignments to a variable, or using a variable with the post-increment operator. The C and C++ standards state that a variable that is subject to a side-effect should not be used more that once in an expression. As an example, this statement violates that rule:

```
*d++ = *d;
```

Simulator
A debugging tool that runs on the host and behaves as similar to the target processor as possible. A simulator is used for debugging the application when the hardware is unavailable, or not needed for proper debugging. A simulator is usually not connected to any physical peripheral devices. A simulated processor is often slower, or even much slower, than the real hardware.

Single stepping
Executing one instruction or one C statement at a time in the debugger.

State machine diagram
A graphical representation of your state machine model, or parts of your model.

State machine file
A file that contains the state machine diagram for the designed state machine model. The state machine file represents a way of modularizing a Visual State system. When a system is split into more than one state machine file, it is possible to gain the benefits of team development on the same system. The filename extension is `vsr`.

State machine model
The state machine part of your application as designed with IAR Visual State.

State machine template
The design of (or part of) a state machine model that can be reused. The template can contain states, regions, elements, and transitions. The template can even itself refer to another state machine template. A state machine template can be used at any level in the design except at the top. It is stored in a file with the filename extension `vssm`. Compare *Submachine state*.

Static object
An object whose memory is allocated at link-time and is created during system startup (or at first use). Compare *Dynamic object*.

Statically allocated memory
This kind of memory is allocated once and for all at link-time, and remains valid all through the execution of the application. Variables that are either global or declared `static` are allocated this way.

Structure value
An umbrella term for structs and unions. A struct is a collection of data object placed sequentially in memory (possibly with pad bytes between them). A union is a collection of data sharing the same memory location.

Submachine state
A state that can refer to a specific state machine template, to provide a concrete instance of the state machine template. As part of the submachine state, the parts of the state machine template meant to be specified when used, must be bound. Compare *State machine template*.

Substate
A state that is below another state in the state machine diagram.

Superstate
A state that in itself contains one or more state machines.

System / Visual State system
A collection of one or more top-level state machines, and their files (filename extension `vsr`). If top-level state machines are grouped in the same system, they can be synchronized to each other via state conditions. The system is the logical unit of a state machine model. Thus, when an event occurs, it is interpreted on a per system basis. Compare *Top-level state machine*.

T

Target
1. An architecture. 2. A piece of hardware. The particular embedded system you are developing the application for. The term is usually used to distinguish the system from the host system.

Task (thread)
A task is an execution thread in a system. Systems that contain many tasks that execute in parallel are called multitasking systems. Because a processor only executes one instruction stream at the time, most systems implement some sort of task-switch mechanism (often called context switch) so that all tasks get their share of processing time. The process of determining which task that should be allowed to run next is called scheduling. Two common scheduling methods are *Preemptive multitasking* and *Round Robin*.

Tentative definition
A variable that can be defined in multiple files, provided that the definition is identical and that it is an absolute variable.

Timer
A peripheral that counts independent of the program execution.

Timeslice
The (longest) time an RTOS allows a task to run without running the task-scheduling algorithm. A task might be allowed to execute during several consecutive time slices before being switched out. A task might also not be allowed to use its entire time slice, for example if, in a preemptive system, a higher priority task is activated by an interrupt.

Top-level state machine
The topmost state in a state hierarchy determines the top-level state machine. Such a state cannot be nested, they can only be building blocks right below a system. They are stored in files with the filename extension `vsr`. Compare *System / Visual State system*.

Translation unit
A source file together with all the header files and source files included via the preprocessor directive `#include`, except for the lines skipped by conditional preprocessor directives such as `#if` and `#ifdef`.

Transition element
The non-graphical elements available in IAR Visual State and which you can use when defining conditions and actions for transitions and state reactions.

You create transition elements in the scope of top-level state machines, projects, state machine templates, or element files.

Transition element file
A file that contains transition elements and nothing else. Having transition elements in element files allows you to reuse small blocks of transition elements where you want. Transition element files are similar to include files in the C/C++ programming language. Adding an element file makes the transition elements in the files defined and available where you added them. Transition element files have the filename extension `.vste`.

Trap
A trap is an interrupt initiated by inserting a special instruction into the instruction stream. Many systems use traps to call operating system functions. Another name for trap is software interrupt.

Type qualifiers
In Standard C/C++, `const` or `volatile`. IAR Systems compilers usually add target-specific type qualifiers for memory and other type attributes.

V

Volatile storage
Data stored in a volatile storage device is not retained when the power to the device is turned off. To preserve data during a power-down cycle, you should store it in non-volatile storage. This should not be confused with the C keyword `volatile`. Compare *Non-volatile storage*.

von Neumann architecture
A computer architecture where both instructions and data are transferred over a common data channel. Compare *Harvard architecture*.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Symbols