

Getting Started

with IAR visualSTATE®

COPYRIGHT NOTICE

© Copyright 2009 IAR Systems AB.

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

DISCLAIMER

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

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

TRADEMARKS

visualSTATE, IAR Systems, IAR Embedded Workbench, and the IAR Systems logotype are trademarks or registered trademarks owned by IAR Systems AB.

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

Microsoft and Windows are registered trademarks of Microsoft Corporation.

All other product names are trademarks or registered trademarks of their respective owners.

EDITION NOTICE

First edition: October 2009

Part number: GSVS-1

This guide applies to version 6.3 and later of IAR visualSTATE®.

Internal reference: IJOA

Contents

| | |
|---|----|
| Preface | 7 |
| Who should read this guide | 7 |
| Other documentation | 7 |
| Online help | 7 |
| Recommended web sites | 8 |
| Document conventions | 8 |
| Typographic conventions | 8 |
| Naming conventions | 8 |
| Introduction | 11 |
| The toolset | 11 |
| Important features and advantages | 11 |
| High-level design | 11 |
| Automatic code generation from a design model | 12 |
| Simulation/validation on a design model | 12 |
| Formal model checking on a design model | 12 |
| Model debugging on target hardware | 13 |
| UML (Unified Modeling Language) | 13 |
| Natural interrupt handling | 14 |
| Easy integration with an RTOS | 14 |
| Prototyping a design before having the hardware | 14 |
| Asynchronous event handling | 14 |
| Basics and concepts | 15 |
| Control logic vs. data manipulation and device drivers | 15 |
| Code required for an application | 16 |
| The state machine model | 17 |
| States | 17 |
| Events | 18 |
| Transitions | 18 |
| Actions | 19 |

| | |
|--|----|
| Supported expression syntax | 19 |
| Primitives to express guard conditions on transitions | 19 |
| State machine hierarchy | 20 |
| Terminology | 20 |
| Getting started | 23 |
| The development cycle | 23 |
| Hierarchy | 23 |
| Project examples | 25 |
| Creating a workspace | 25 |
| Setting options | 26 |
| Specifying the output directory for generated code | 27 |
| Reloading files in the Navigator | 27 |
| Designing your model | 29 |
| Start using the Designer | 29 |
| Create your first statechart | 30 |
| Create a state | 30 |
| Create a transition | 30 |
| Create an initial state | 32 |
| Create an action function to use with a transition | 32 |
| Composite states | 34 |
| Create a composite state consisting of concurrent regions | 34 |
| Create a composite state consisting of mutually exclusive sub- states | 34 |
| Some statechart diagram examples | 35 |
| Transition triggered by external events | 35 |
| Transition guarded by a Boolean condition | 35 |
| Entry and exit reactions | 36 |
| Testing your model | 37 |
| Validating your model | 37 |
| Graphical simulation | 37 |
| Testing your model in the target application | 37 |
| Verification | 38 |

| | |
|---|----|
| Verifying your model | 38 |
| Tracing your project | 39 |
| C code needed for your model | 41 |
| Generating code from statecharts | 41 |
| Completing your application | 41 |
| Coder-generated code | 41 |
| User-written code | 42 |
| Integrating the C files | 42 |
| Example of code for event handling | 43 |
| Example of visualSTATE API code | 44 |
| Execution | 45 |
| Understanding the visualSTATE control logic code | 46 |
| Table-based code generation | 46 |
| Readable code generation | 48 |
| Additional features | 51 |
| Documenting your project | 51 |
| Prototyping | 52 |

Preface

WHO SHOULD READ THIS GUIDE

You should read this guide if you want to get started using visualSTATE® without first having to read all the reference information, included in PDF format. You should have working knowledge of:

- The C programming language
- Basic principles of state/event modeling
- Application development for embedded systems
- The operating system of your host computer.

Refer to the *visualSTATE Reference Guide*, the *IAR visualSTATE*, and the *visualSTATE Concept Guide* for detailed information about the development tools incorporated in visualSTATE.

OTHER DOCUMENTATION

The visualSTATE development tools are described in a series of guides. For information about:

- Installing the visualSTATE suite, refer to the *visualSTATE Installation Guide*
- The basic concepts and philosophy behind visualSTATE, refer to the *visualSTATE Concept Guide*
- Hands-on tutorials of using visualSTATE, refer to the *visualSTATE Quick Start Tutorial*
- Information about the menus, dialog boxes and windows available in the visualSTATE graphical user interface, refer to the *visualSTATE User Guide*
- Detailed reference information about the constructs, elements, and principles of state machine modeling, refer to the *visualSTATE Reference Guide*
- Using the visualSTATE APIs, refer to the *visualSTATE API Guide*.

All of these guides are delivered in hypertext PDF format on the installation media. Note that additional documentation might be available from the **Help** menu depending on your product installation.

Online help

The visualSTATE Navigator, Designer, and Validator applications offer online help. Open the **Help** menu or press the F1 key to access the online help.

To display online help for options in the **Navigator Settings** dialog box and Navigator **Project>Options** dialog boxes, select the option and right-click, or press Shift+F1.

Recommended web sites



- The IAR Systems web site, **www.iar.com**, holds application notes and other product information.
- The web site of the Object Management Group, **www.uml.org**, contains UML specifications, articles about UML, and other resources.

DOCUMENT CONVENTIONS

When referring to a directory in your product installation, for example `visualSTATE 6.n\Doc`, the full path to the location is assumed, for example `c:\Program Files\IAR Systems\visualSTATE 6.n\Doc`.

Typographic conventions

This guide uses the following typographic conventions:

| Style | Used for |
|--|--|
| <code>computer</code> | <ul style="list-style-type: none">• Source code examples and file paths.• Binary, hexadecimal, and octal numbers. |
| bold | Names of menus, menu commands, buttons, and dialog boxes that appear on the screen. |
| <i>italic</i> | <ul style="list-style-type: none">• A cross-reference within this guide or to another guide.• Emphasis. |
| <code><filename>.<ext></code> | A file generated by visualSTATE Coder. |
|  | Helpful tips and programming hints. |
|  | Warnings. |

Naming conventions

The following naming conventions are used for the products and tools from IAR Systems® referred to in this guide:

| Brand name | Generic term |
|---------------------------|--------------|
| IAR visualSTATE | visualSTATE |
| IAR visualSTATE Navigator | Navigator |
| IAR visualSTATE Designer | Designer |

| Brand name | Generic term |
|-----------------------------|---------------------|
| IAR visualSTATE Verificator | Verificator |
| IAR visualSTATE Validator | Validator |
| IAR visualSTATE Coder | Coder |
| IAR visualSTATE Documenter | Documenter |
| IAR visualSTATE Basic API | Basic API |
| IAR visualSTATE Expert API | Expert API |
| IAR visualSTATE Expert DLL | Expert DLL |
| IAR visualSTATE project | project |
| IAR visualSTATE system | system |

Introduction

THE TOOLSET

visualSTATE® is a toolset for design, test, formal verification (model checking), code generation, and high-level hardware debugging of state/event systems in general and state machines in particular. visualSTATE is based on the state machine subset of UML (Unified Modeling Language), which combines the Mealy and Moore notations with the concept of hierarchy and concurrency.

visualSTATE consists of these fully integrated tools:

Navigator: A project management tool for the overall handling of visualSTATE projects, from model design over test and simulation to code generation and documentation of projects. With the Navigator you access and activate the other modules of the visualSTATE software, and set options for the Verificator, Coder and Documenter.

Designer: An application for designing statechart diagrams using the UML notation.

Verificator: A test tool for dynamic formal verification of models created with the Designer.

Validator: An application for simulating, analyzing, and debugging models created with the Designer. With the Validator you can test the functionality of your design. Using the RealLink facility, you can test visualSTATE models in a target application, and using the C-SPYLink plug-in, you can perform high-level state machine debugging of visualSTATE applications in the IAR Embedded Workbench® C-SPY Debugger.

Coder: The Coder can automatically generate code on the basis of models created with the Designer. The automatically generated code must be combined with a visualSTATE application programming interface (API) and manually written code.

Documenter: A tool for creating an up-to-date documentation report on your project, including design, tests, and code generation.

IMPORTANT FEATURES AND ADVANTAGES

High-level design

Graphical design tools can be cumbersome to work with as they force you to specify even very simple things, but the UML state machine subset lets you design state machines at the right abstraction level. Because the UML state machine

subset incorporates hierarchy and 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.

The high level of design also makes it easier to discuss the control logic on the design model level with non-programmers, such as product managers.

Automatic code generation from a design model

By automatically generating the code for your state machine logic, you avoid the hand coding of complicated `switch` and `if` statements, or the restrictions imposed by a function pointer table approach.

The generated code makes no assumptions about any compiler-specific features except ISO/ANSI conformance, and uses no construct that is not fully ISO/ANSI-conformant.

You can configure the visualSTATE Coder to use compiler-specific keywords to place state machine code and data in 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 Coder in many different ways to balance the needs of the MCU target, the compiler, and coding standards.

The code generated by visualSTATE focuses on the control logic of a state machine system. This part of the code should not be modified by hand, for several reasons of which the most important is 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.

Simulation/validation on a design model

With design level simulation, you can start testing your solution as soon as you have saved the very 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 on a design 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 test 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 visualSTATE you can debug on target hardware with feedback directly in the design diagrams—to see exactly which state configuration is active and which transition was taken to enter the state configuration.

If you also use IAR Embedded Workbench you can choose to use the C-SPYLink facility to get high-level design model feedback directly in the IAR C-SPY Debugger. C-SPYLink includes graphical animation in the state diagram when it executes, the possibility to set breakpoints at state machine level instead of C level, and trace and log functionality.

If you use a different build chain or cannot use a hardware debug solution with C-SPY, you can use the RealLink facility to communicate state machine data over a separate communication channel, for example, an RS232 port.

Support for high-integrity systems

visualSTATE is suited for many design tasks involving 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.

By designing a system of state machines using visualSTATE, you can take advantage of the formal verification to find issues in your design that are virtually 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 *Verification*, page 38.

UML (Unified Modeling Language)

The UML notation for state machines is used in visualSTATE. This notation is based on hierarchical state machines, and concurrency can be used at any level in the state 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. Some memory can be applied to the state machines.

You can read more about the UML concepts in the *OMG Unified Modeling Language Specification*, version 2.3, February 2009, available from www.omg.org.

Natural interrupt handling

The visualSTATE runtime execution engine deals with events—abstractions of occurrences in the environment. This makes it natural to map an interrupt to a visualSTATE event, if the interrupt should have influence on the state machine.

A typical visualSTATE application runs the state machine engine as part of the main loop, if there is an event to process.

Exactly how the interrupt routine communicates with the state machine engine is up to you. But implementation methods range from letting the interrupt routine set a flag that the main loop can detect, 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.

Easy integration with an RTOS

Use visualSTATE 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 visualSTATE code to run in different tasks, split the state machines into different systems. A visualSTATE 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 house more than one system at a time.

Prototyping a design before having the hardware

You can easily integrate code generated by visualSTATE with an application developed using a RAD tool like Altia Design, Microsoft® Visual Basic®, Microsoft® Visual C++®, Borland® Delphi™, or any other GUI toolkit of your choice.

Asynchronous event handling

Asynchronous events are handled if they are forwarded to the visualSTATE 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 visualSTATE control logic.

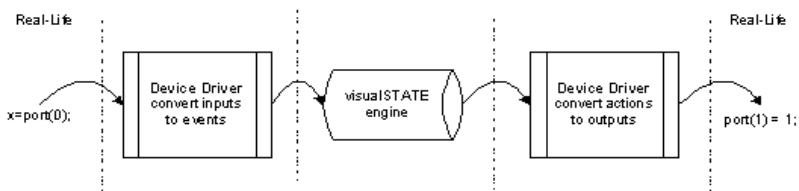
Basics and concepts

CONTROL LOGIC VS. DATA MANIPULATION AND DEVICE DRIVERS

An embedded application is typically a combination of control logic, data manipulation, and device driver code.

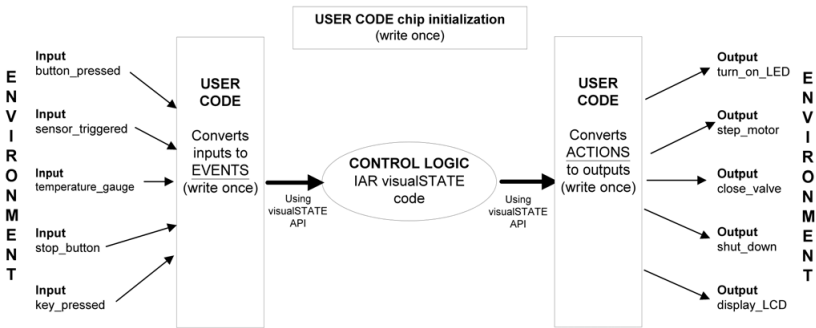
Typically, device drivers for a specific target processor are written only once. They might then become part of a library which remains more or less constant from project to project. However, the control logic part that implements the features and specification of a given product might change dramatically from project to project.

Using visualSTATE®, you can develop control logic for event-driven systems based on state machines, where events input 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.



- 1 The externally generated input is processed by the device driver, by way of interrupts or polling.
- 2 The driver informs the visualSTATE 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 that use device drivers for output can be called.

This figure summarizes the parts that visualSTATE handles in an embedded application:



CODE REQUIRED FOR AN APPLICATION

Piecing your application together with visualSTATE® as your main control logic engine is easy. Basically, you still have full control over the structure of your application code. You integrate the code created by visualSTATE into the application by calling the appropriate visualSTATE API functions.

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

- visualSTATE Coder-generated code
- visualSTATE API code
- Code written by you for *event preprocessing*, *event queues*, *device drivers*, *action functions*, and code for calling the functions in the visualSTATE API.

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

Two types of Coder-generated code exist:

- Table-based code, which is extremely compact
- Human-readable C code.

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.

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

In other words, to create a final embedded application using visualSTATE-generated code, you must:

- Manually write code for event preprocessing (device drivers), event queues (if needed), and action functions
- Integrate the code written by you with the Coder-generated code by means of the visualSTATE API, a set of files that provide an interface between the two types of code.

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

THE STATE MACHINE MODEL

A state machine is an abstraction of an application, created by drawing statechart diagrams. A state machine has several important design advantages, being a visual representation which is easy to create, understand, communicate, and change.

Using statecharts allows you to develop the specification and application hand-in-hand in a natural, iterative fashion.

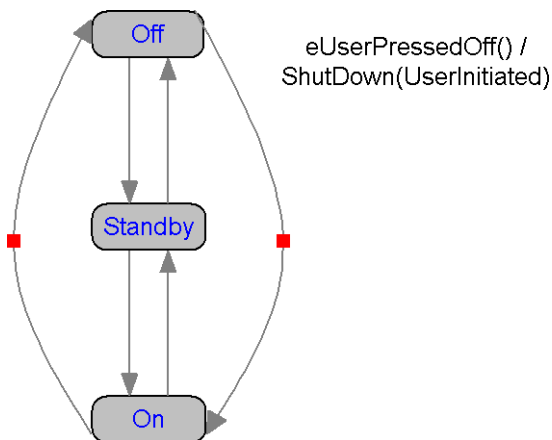
The state machine model is widely used to describe discrete systems, where the current behavior is a result of previously occurring events. At any given point in time, the system is in one of several possible states. The system can change states depending on input from the environment. As a state change occurs, actions can be performed on the environment.

A state machine is generally defined by a finite set of:

- States
- Events
- Transitions
- Actions
- Primitives to express guard conditions on transitions.

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. An electronic device subsystem can, for example, be On or Off, a door can be Open, or it can be ClosedAndUnlocked, etc. 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. In a state machine diagram, a state is usually drawn as a rectangular shape, or a circle. In visualSTATE an ordinary state is drawn as a rectangle or a square. There is also a set of states with a special meaning that are drawn as circles.



Events

An event is an input message to the state machine. An event is typically something that happens in the outside environment that the state machine logic must know. An event can cause the state machine to change states and to perform an action of some sort, but only if the state machine is in a state where the event has meaning. For example, an event called TurnOn is probably not meaningful if a device is already in the On state, but has meaning if the state machine is in the Standby state. Thus, a state machine event is an abstraction of one or more real-world events or messages.

Transitions

A transition is a change from one state to another, usually triggered by an event. Transitions are drawn as directed arcs, labeled with an event. If the state machine is in the start state of a transition and the event that is labeled on the transition occurs, the state machine will change states to the destination state.

In visualSTATE, a transition must always have a trigger, implicitly or explicitly.

The description of a visualSTATE transition is divided into a condition side and an action side. The condition side of a transition describes which conditions must be satisfied for the action side to be executed. The action side of a transition describes all the actions that will be executed if the conditions on the condition side are satisfied.

The condition and action sides are separated by a slash (/), for example:

```
E1() [(x==0)] A !B / [x=A1()] A2() D^S1
```

The condition side is to the left of the slash and action side of the transition is to the right of the slash.

Actions

An action is an activity to be performed by the state machine at a given point in time. An action is something that the state machine must perform on the environment. It can be a simple expression or a complete function. Actions are often used for manipulating hardware. Actions are associated with transitions or with entering or exiting a specific state.

Supported expression syntax

A visualSTATE expression can be any valid C expression of the C operators, identifiers, floating-point and integer constants, with these limitations:

These operators are *not* supported:

- . (member)
- > (member by pointer)
- * (indirection (dereference))
- (type) (cast)
- sizeof
- ?: (ternary)
- , (comma)

These elements of the C syntax are *not* supported either:

- long, double, and float constants
- suffixes for integer constants
- octal integer constants
- multiple assignments or increments/decrements in the same expression.

These limitations also apply:

- An assignment and increment/decrement operator must be placed first in the expression, not in the middle.
- Event arguments of void* type can only be passed to action function arguments.

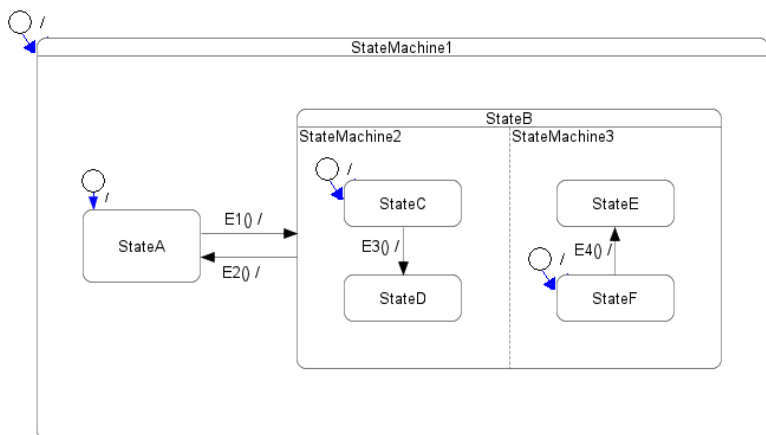
Primitives to express guard conditions on transitions

A guard condition is a Boolean expression that must be true for the transition to be taken. The fact that Boolean expressions can be used on transitions implies that you can somehow manipulate data in the state machine as well, and in a

visualSTATE design you can create and manipulate variables of integer types and floating-point types to extend the expressiveness of the state machine.

State machine hierarchy

visualSTATE is 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*.



If a superstate is composed of mutually exclusive substates, it corresponds to one state machine. If the superstate is composed of concurrent regions, each region corresponds to a state machine. Each of these state machines can contain other state machines.

TERMINOLOGY

You should be familiar with some of the most commonly used terms in visualSTATE. These terms are listed in the table below. You can read more about the individual concepts in the *visualSTATE Reference Guide*.

The Navigator workspace is a graphical environment for handling a collection of projects, systems, and statechart files. The workspace contains session-specific information.

A project is a collection of systems. Each project is capable of containing several statecharts. The project also contains global elements. The project data is stored in a project file which has the filename extension `vsp`.

A system is a collection of one or more statechart files. If state/event models are grouped in the same system, they can be synchronized to each other via state conditions. Thus, their behavior can be affected by the behavior of the other state/event models within the same system.

Statechart files contain the logic information about the designed model, for example states and transitions. The statechart file represents a way of modularizing a visualSTATE system. When a system is split into more than one statechart file, it is possible to gain the benefits of team development on the same system. The filename extension is `.vsr`.

Superstates are states that in themselves contain one or more state machines.

A topstate is the topmost state in a state hierarchy.

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

Regions define concurrent subsystems and represent hierarchical state machines.

A topstate region is a region within the topmost concurrent state.

A device driver controls a device and acts like a translator between the device and programs that use the device. Each device has its own set of specialized commands that only its driver knows. In contrast, most programs access devices by using generic commands. The driver, therefore, accepts generic commands from a program and then translates them into specialized commands for the specific device, and vice versa. A device driver can control anything from a LED or a hardware timer to a mass storage device or a wireless communications device, etc.

API calls are function calls to routines contained in the API (application programming interface) provided with visualSTATE.

Local elements are events, actions, variables, signals etc. that are defined at topstate level. They normally have the scope of the topstate itself.

Global elements are events, actions, variables, signals etc. that are defined at project level. Thus, they have the scope of the visualSTATE project, including all systems contained in it.

Getting started

When you have installed visualSTATE®, you are ready to get started. For installation, refer to the visualSTATE Installation Guide.

THE DEVELOPMENT CYCLE

The typical development cycle looks like this:

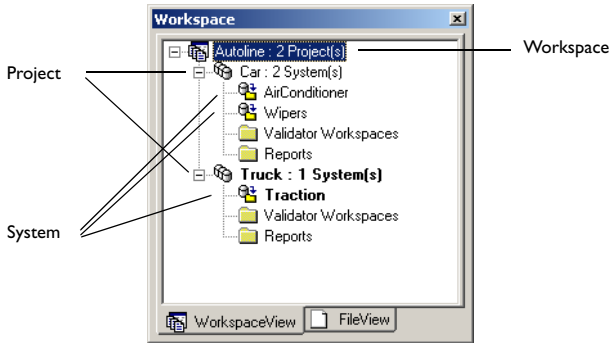
- 1 Draw up the overall structure of your application
- 2 Design models of state machines in the Designer
- 3 Test your model:
 - Simulate, analyze, and debug the model using the Validator
 - Verify the logic of the model using the Verifier.
- 4 Document your project. This means that you create a report using the Documenter.
- 5 Use the IDE of the C compiler you are using—for example, the IAR Embedded Workbench IDE—to create a project that includes all the necessary C files
- 6 Generate the C code for your model. On the target, the code will behave exactly as the model you designed.
- 7 Compile and link your C files using the IDE of your C compiler, and debug the application.

You can also monitor and control the runtime behavior of a visualSTATE model in a target application, using either the Validator RealLink utility or in IAR Embedded Workbench using C-SPYLink.

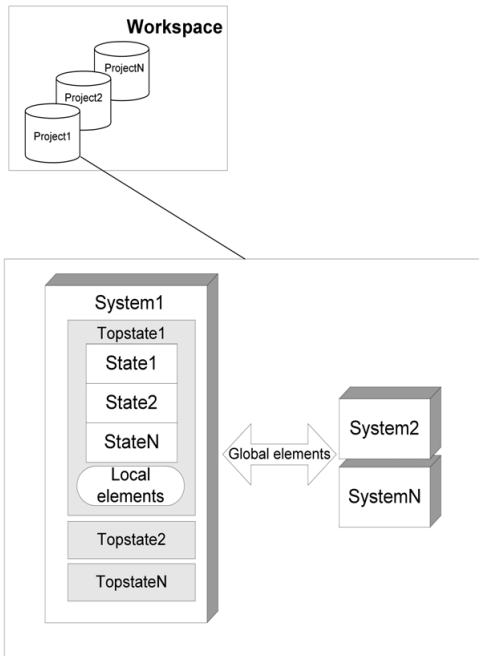
HIERARCHY

visualSTATE application development uses multiple levels of hierarchical representation to better structure and manage the overall design.

- | | |
|--------------------------------|-------------------------------------|
| ● Navigator workspace (.vnrw): | Can contain any number of projects |
| ● Project (.vsp): | Can contain any number of systems |
| ● System (no file): | Can contain any number of topstates |
| ● Statechart file (.vsr): | Can contain any number of states |



The visualSTATE hierarchy can be visualized as shown in this figure:



- Each project has one Global Elements section
- Each topstate has one Local Elements section.

Element sections contain the names of events, actions, variables, etc.

PROJECT EXAMPLES

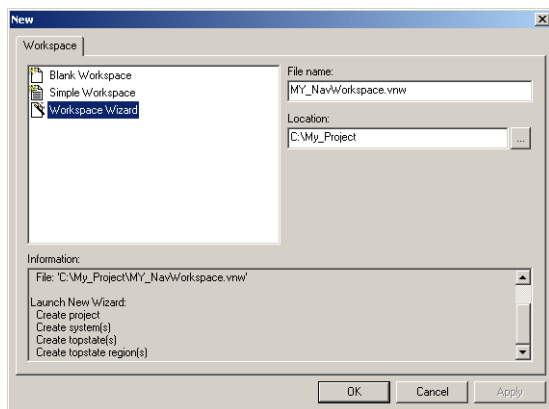
The visualSTATE software package includes examples of application designs created with visualSTATE.

To open the examples, choose

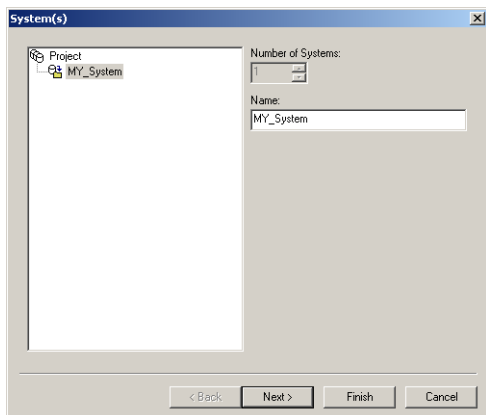
Start>IAR Systems>visualSTATE 6.n>Examples, or open them from within the Navigator—browse for the `\IAR Systems\visualSTATE 6.n\Examples` directory.

CREATING A WORKSPACE

- 1 Launch visualSTATE Navigator from the Windows **Start** menu.
- 2 Click **Create a New Workspace** and **OK** to open the **New** dialog box.
- 3 Select **Workspace Wizard** and type a filename for your Navigator workspace. Specify a directory to save it to and click **OK**.



- 4 In the **System(s)** dialog box that is displayed, select the system, type a name for it, and click **Finish**:



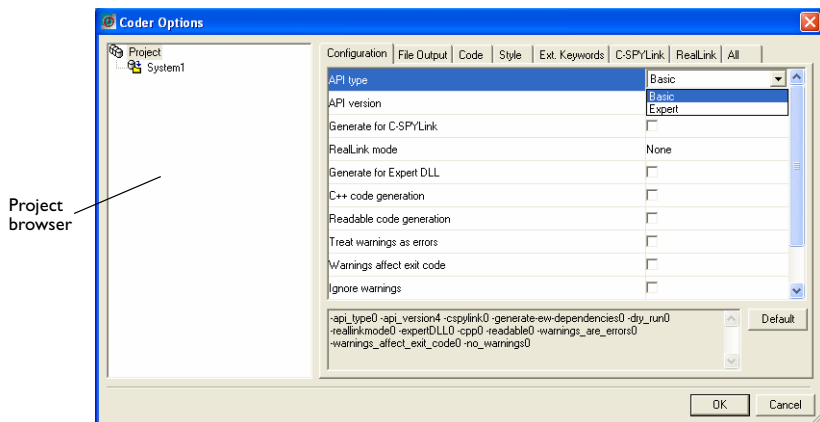
A status window shows which files will be created. Click **OK**. The Designer is launched.

You have now created a Navigator workspace with a project that contains one system with the topstate Topstate1.



SETTING OPTIONS

- 1 Choose **Project>Options** and the tool that you want to set options for.

An options dialog box is displayed:



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

- 2 In the Project browser, select the project or system that you want to set options for.
- 3 Click the tab containing the category of options you want to set. To view all options available, click the **All** tab.
 - Some options have a drop-down list box. Click in the list box and select the appropriate value.
 - Some options have check boxes. Click the option check box to select or deselect the option.
 - Some options have a text box where you can type the value for the option.
 - Some options have buttons that are displayed when you select the option:
 -  Click this button to browse for files to use
 -  Click this button to display a pop-up menu.

The selected values are shown as command line options in the display area below the options list.



If you right-click on any option name, a description of the option will be displayed.

Some options cannot be changed, because not all combinations of options are possible. To restore the options to their default values, click the **Default** button.

Specifying the output directory for generated code

- 1 In the Navigator Workspace window, right-click on your project and choose **Options>Code generation** to open the **Coder Options** dialog box.
- 2 On the **File Output** page, specify an output directory (**Output path**) for the generated C code. The default location is the `coder` directory in the directory that contains the project file (`.vsp`).
- 3 On the **Code** page you can select the **Generate digital signature** option, to secure the integrity of the model and guard against accidentally mixing different versions of user code and model code.

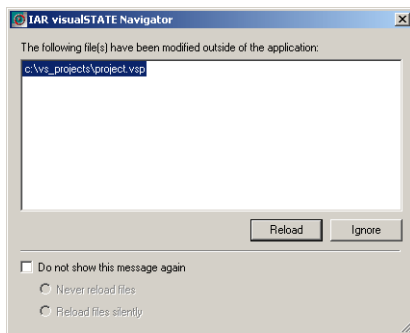
Click **OK**.

RELOADING FILES IN THE NAVIGATOR

By default, you will receive a reload message in the Navigator when the project files or statechart files (`*.vsp` and `*.vsr` files) in the current workspace have been modified outside the Navigator.

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

To change the way the reload message is displayed, choose **Tools>Settings** and change the setting for the option **Automatic file reload**.

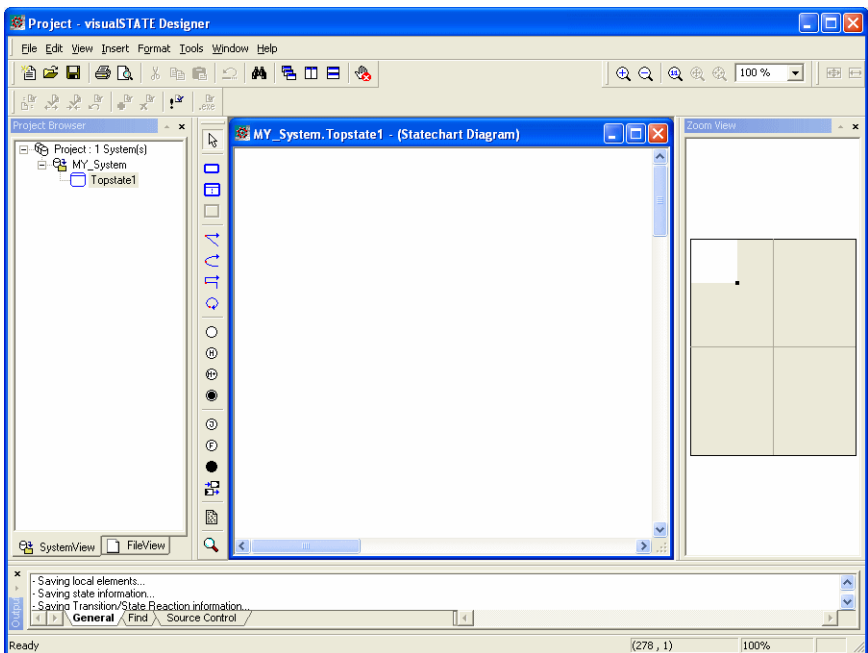


Designing your model

This chapter gives examples of how to design the program logic using statecharts.

START USING THE DESIGNER

- 1 In the Navigator Workspace window, select your project and choose **Project>Designer** to launch the Designer application.
- 2 In the Designer Project Browser, double-click the topstate to open the System View window.
- 3 Your project has several levels of hierarchy. Double-click the topstate rectangle in the System View to open the Statechart Diagram window.



A topstate is the “root state” for a statechart diagram. A statechart diagram can in turn contain any number of hierarchic and parallel states.

Almost all of the tools on the Diagram toolbar should now be available. If you hold your mouse pointer over a tool icon, a description is displayed in the status bar at the bottom of the window.

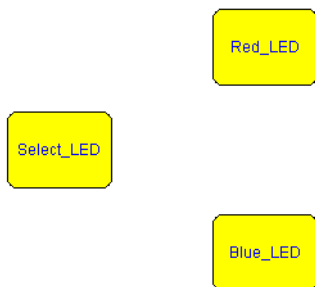
To activate a tool, click on it. To deactivate a tool, right-click in the window.

CREATE YOUR FIRST STATECHART

This section describes the basic procedures involved when you create a statechart.

Create a state

- 1 Click the Simple State button on the Diagram toolbar and click in the Statechart Diagram window to create a state. Right-click in the window to deactivate the tool. You can resize and move the state you have drawn as necessary.
- 2 To give a state a name, click on the default state name and type the new name of the state. Create three simple states and call them, for example, `Select_LED`, `Red_LED`, and `Blue_LED`:



Create a transition



Click the Transition button on the Diagram toolbar. In the Statechart Diagram window, click on the desired source state, move the pointer to the destination state and click to complete the transition. Right-click in the window to deactivate the tool.

The three different Add Transition tools allow you to draw straight, curved, and orthogonal lines.

To move a transition:

- 1 Select the transition and move the cursor to one of its end points.
- 2 Drag the point to the right position, release the mouse button, and click again.

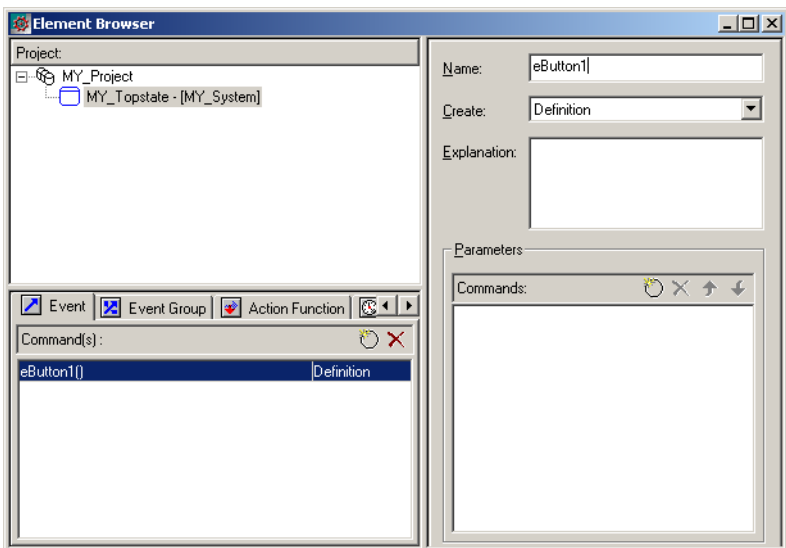
To add a *route point* to a transition:

- 1 Select the transition and move the pointer to an existing route point or end point and Ctrl-click.
- 2 Drag the pointer to the desired location where you release the mouse button and the Ctrl key.

A route point is an anchor that you drag to change the look of the transition.

To add an event to a transition:

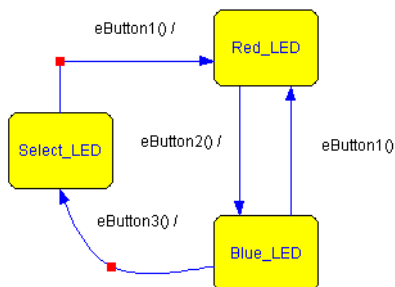
- 1 Choose **View>Element Browser** and select the topstate in the Element Browser.
- 2 Click the **Event** tab and click the **New** button in the Element Browser:



In the pane to the right, give the event a name and optionally a description in the **Explanation** field. Give the event the name `eButton1`.

Return to the Statechart Diagram window. Double-click on the text box of the transition you wish to compose. This displays the **Compose Transition** dialog box. Click on **Trigger** in the **Rule** list. Then double-click the event in the **Element** list that you want to add to the trigger and click **OK**.

For the purposes of our example, create three more transitions and two more events (eButton2 and eButton3). Add route points to two of the transitions, so that your statechart diagram looks like this:

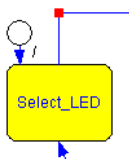


Create an initial state



Click the Initial State button on the Diagram toolbar.

Add an initial state to the state chart. Also add an empty transition from the initial state to one of the simple states (for example, the state `Select_LED`).

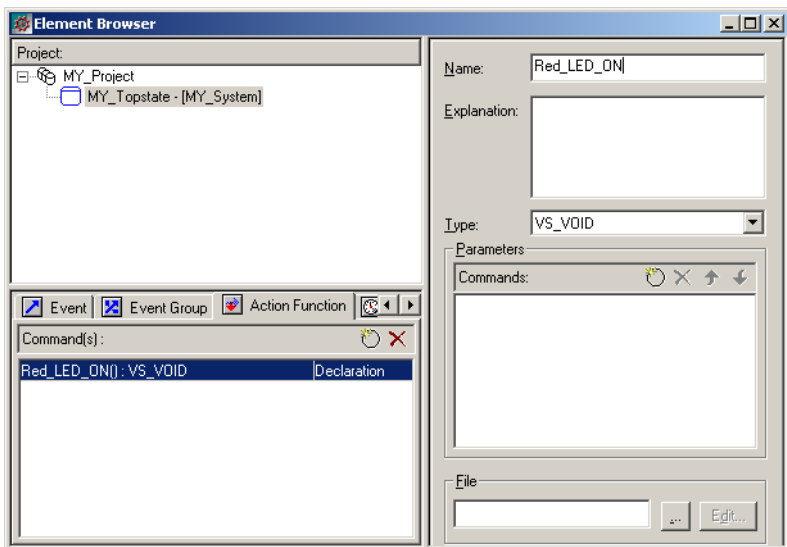


Create an action function to use with a transition



Choose **View>Element Browser** and click the **Action Function** tab. Then click the **New** button.

In the pane to the right, give the action function a name and optionally a description in the **Explanation** field. Call it, for example, `Red_LED_ON`.

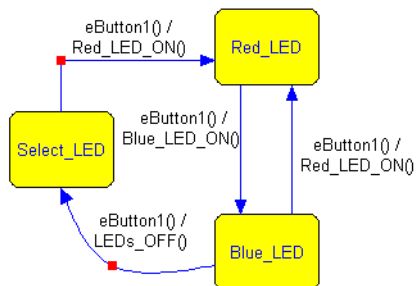


Return to the Statechart Diagram window. Double-click on the text box that belongs to the transition between the `Select_LED` state and the `Red_LED` state. In the **Compose Transition** dialog box, click on **Action Expression** in the **Rule** list. Then double-click the action function in the **Element** list that you want to add to the transition and click **OK**.

Create action functions called `Blue_LED_ON` and `LEDs_OFF` for the other transitions, so that your statechart diagram looks like this:

When you have designed your statechart, save the project and exit the Designer.

Click **Yes** when prompted to reload files in the Navigator.



COMPOSITE STATES

A *composite state* consists of several concurrent regions, or mutually exclusive substates. A *region* is an area that holds states. Regions are used in states and topstates to define concurrent subsystems and represent hierarchical state machines.

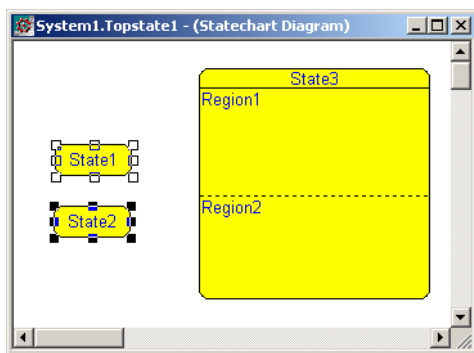
Create a composite state consisting of concurrent regions



- 1 Click the Composite State button on the Diagram toolbar.
- 2 Click in the Statechart Diagram window to create a state with one region and then right-click in the window to deactivate the tool.
- 3 To add a region, right-click in the region and choose **Insert Region** from the context menu.

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.

To insert existing states in a concurrent region, use the selection tool (the arrow tool) to select the states you want to move. Then drag the states into the region.



Create a composite state consisting of mutually exclusive substates

- 1 Select the states that you want to turn into substates.
- 2 Drag the states onto the state intended to be the composite state. A region is automatically inserted, indicated by a horizontal line below the state name.

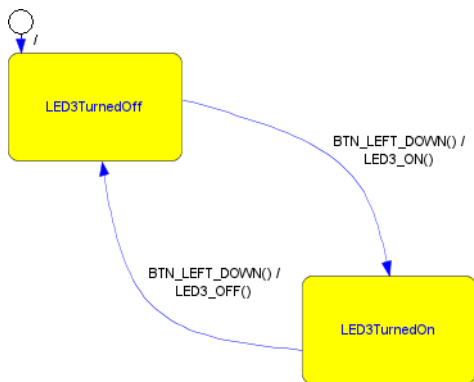
A composite state composed of mutually exclusive substates can be changed to a composite state with concurrent regions just by adding a region. This will

automatically create two regions and move the original substates into one of the regions.

SOME STATECHART DIAGRAM EXAMPLES

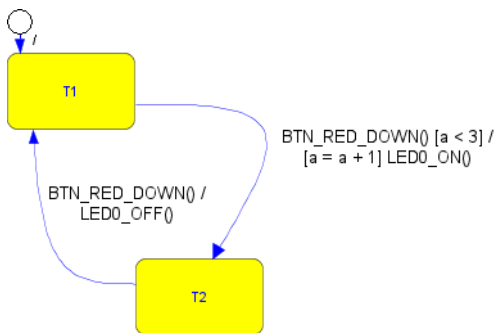
Transition triggered by external events

When the LEFT button is pressed, the transition from state LED3TurnedOff to LED3TurnedOn is triggered and accompanied by the action LED3_On. Then, if the LEFT button is pressed again, the transition from LED3TurnedOn to LED3TurnedOff is triggered and LED3 is turned off.

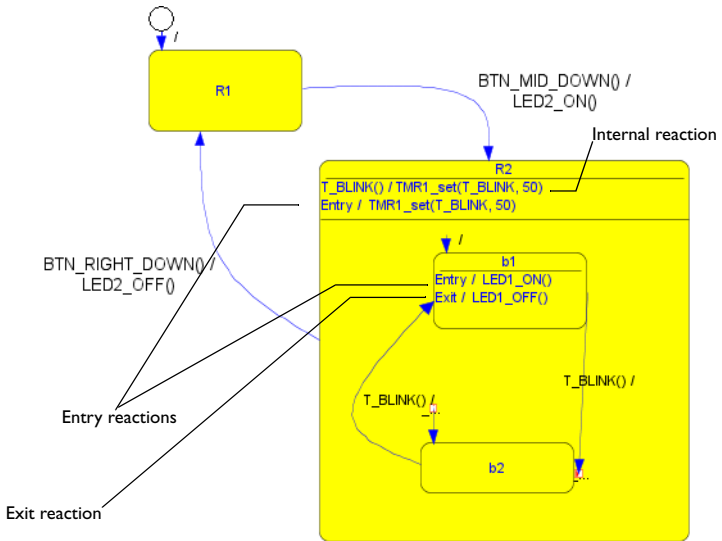


Transition guarded by a Boolean condition

When the RED button is pressed, the transition from state T1 to T2 is triggered and accompanied by two actions. LED0 is turned on, and the variable a is incremented. This transition is also guarded by the condition $a < 3$. That expression must evaluate to TRUE for the transition to occur. In this case, LED0 will only be turned on three times. After that, the condition $a < 3$ evaluates to FALSE and the transition will not fire when the RED button is pressed.



Entry and exit reactions



Pressing the middle button (`BTN_MID_DOWN`) causes a transition from state `R1` to `R2`. This transition causes two entry reactions to be executed. When entering superstate `R2`, the timer action function `TMR1_set` is initialized to count down 50 system ticks. As this occurs, the default substate `b1` is activated and its entry reaction `LED1_ON` is executed.

After 50 ticks, `TMR1_set` times out and the event `T_BLINK` is sent. This causes two transitions to fire. First the internal reaction of `R2` is triggered, resulting in `TMR1_set` beginning another 50-tick countdown. Second, `T_BLINK` triggers the transition from `b1` to `b2` and the exit reaction, `LED1_OFF`, is executed. After the next 50 ticks, `T_BLINK` again causes a transition from `b2` to `b1`, and the entry reaction `LED1_ON` is executed. Thus, `LED1` blinks with an interval of 50 system tick units.

Testing your model

This chapter describes how to test and debug the visualSTATE® model you created in the Designer.

VALIDATING YOUR MODEL

- 1 In the Navigator, choose **Project>Validator**.
- 2 To start the debug process, double-click on `SE_RESET` in the Event window to place your system in its initial state. (When the Validator is started, the only event that you can select is the `SE_RESET` event, to ensure that the system is properly initialized.) You can issue the `SE_RESET` event at any time to force the system back to the initial state.

Double-click events in the Event window to step through your system. In the System window you can see the resulting state combination. In the Action window you can see which actions were generated by the event.

- 3 Choose **Debug>Initialize System** to reset your simulation session.

Graphical simulation

- 1 To view simulation behavior directly in your model, choose **Debug>Graphical Animation** to open the Designer in simulation mode.
- 2 Double-click events in the Event window, to display the resulting state combination with red borders in your statechart. Blue borders are used for the states that were last active.
- 3 When you have debugged your state machine design, exit the Validator and return to the Navigator.

TESTING YOUR MODEL IN THE TARGET APPLICATION

Using either RealLink or the C-SPYLink debug integration with IAR Embedded Workbench feature, you can monitor and control the runtime behavior of your visualSTATE model in the target application.

For detailed information on how to use RealLink, refer to the *visualSTATE User Guide*.

For detailed information on how to use C-SPYLink, see the *visualSTATE C-SPYLink User Guide*.

The Validator can also record the events you trigger and the responses from the state machine system, to so called test sequence files. Such files can be replayed for regression testing and analyzed for static and dynamic model coverage.

VERIFICATION

The Verificator can verify the following properties of a system of interacting state machines:

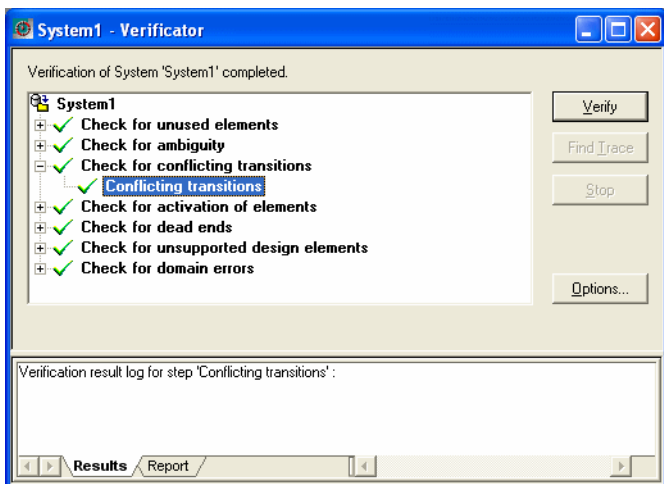
- **Dead-end conditions:** when a state can be entered but never exited, because of some blocking guard condition. This situation can be extremely difficult to test for in a complex state machine.
- **Conflicting transitions:** two or more transitions out from a state that all trigger on the same event and do not have mutually exclusive guard conditions. This is not permitted and will result in a runtime error in the generated code.
- **Unreachable states:** states that cannot be entered by any sequence of events from the environment. A state or set of states that is not reachable is probably an indication that some assumption(s) in the model is wrong.
- **Unused events or signals:** stimuli to the system that are never acted upon
- **Unused transitions:** transitions that will never be taken, regardless of the event sequence fed into the system
- **Unused actions or assignments**
- **Unused variables, parameters, and constants.**

Some of these properties can be partly checked by a simple syntactic check of the original state machine model, for example unused events. It is easy to check whether a certain event is mentioned in the model. However, this check is not enough. An event can be unused in a model even if it is mentioned, if the set of states that acts upon the event is unreachable. To completely answer the question, some form of formal verification must be used.

VERIFYING YOUR MODEL

- 1 Choose **Project>Verify System** in the Navigator to activate the Verificator.
- 2 If the **Verify Warning** dialog box is displayed, select the option **Code generate and verify**.

- 3 View the results of the verification in the Verifier window:



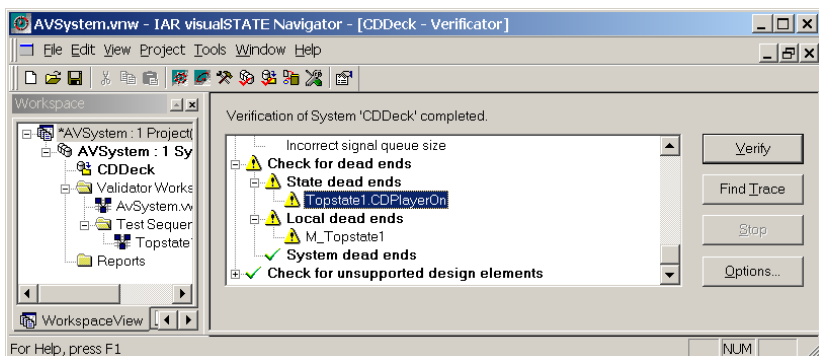
All bold checks are checked, indicating that no errors were found.

- 4 If the verification message contained any errors or warnings, choose **Project>Designer** in the Navigator to return to the Designer and make the necessary changes.

TRACING YOUR PROJECT

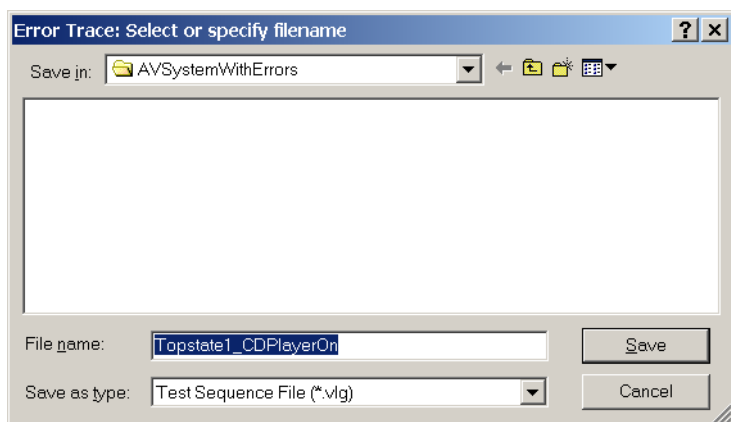
A trace is a sequence of events that will get the system into a desired state configuration. The trace will be saved in a test sequence file. You can only perform a trace in the Navigator if you have just run a verification.

- 1 Select the dead end or conflict you want to trace to.



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

- 3 Select or specify the filename for the trace output file and click **Save**.



- 4 The Navigator will find a trace to the error or warning and save the resulting trace. After the file has been saved, the Validator will be opened with the test sequence file loaded.

For detailed information about the Verifier, refer to the *visualSTATE Concept Guide* and the *visualSTATE User Guide*.

C code needed for your model

This chapter describes how you create the different pieces of C code needed to build your application.

GENERATING CODE FROM STATECHARTS

- 1 In the Navigator, open your project. In the Workspace window, select the project and choose **Project>Code Generate** to activate the Coder.
- 2 The Navigator will display a coder report file where you can see the names of the generated files and their location. This file also contains statistics on the various elements contained in the generated code.

COMPLETING YOUR APPLICATION

To complete your application you need:

- an ANSI C compiler (for your target processor)
- visualSTATE generated code
- user-written code.

- 1 Create a directory structure to collect all the source files needed. For example, create a directory called `Source` with two subdirectories called `Coder-generated` and `User-written`.
- 2 Use the IDE of your C compiler to create a project that includes all the necessary C files, for example, the IAR Embedded Workbench IDE.
- 3 Finally, compile and link your files in the IDE.

Coder-generated code

Place the files created by the visualSTATE Coder in the `Coder-generated` directory.

To direct the Coder to place the generated files in this directory:

- 1 Choose **Project>Options>Code generation** in the Navigator and select your project.
- 2 On the **File Output** page, select the **Output path** field and specify the `Source\Coder-generated` directory.

Note: Never modify the Coder-generated files. If you want to make changes, modify the statechart diagram in the Designer and code-generate the project again.

In this case, these Coder-generated files are located in the directory:

- MY_System.c
- MY_System.h
- MY_SystemAction.h
- MY_SystemData.c
- MY_SystemData.h
- SEMLibB.c/h or MY_SystemSEMLibB.c/h, depending on the setting of the Coder option **Use prefix for API**
- SEMBDef.h
- SEMTypes.h

User-written code

Place the target-specific code that you create in the `User-written` directory.

You must supply the target-specific code (also known as device drivers) that handles the translation of hardware input to `visualSTATE` events, and of `visualSTATE` actions to hardware output. You must also supply a main loop routine that processes the events that occur in the system.

User-written code can be arranged in, for example, these files:

- Actions_to_Output.c
- Input_to_Events.c
- main.c
- simpleEventHandler.c
- eventHandler.h
- LEDsys_Drivers.h
- simpleEventHandler.h

INTEGRATING THE C FILES

To put a complete application together, you must, for example, create:

- A `main.c` file
- Code to initialize your processor
- Code to process inputs
- Code to handle your I/O devices
- Code that you normally need to write, that is not related to the logic of handling a certain event.

You still must create all the “write-once” code. By doing so, you keep full control over the structure of your application and need not adopt a new rigid code structure.

For handling input/events, you must call visualSTATE API functions and pass your event to the visualSTATE system.

You should also create a FIFO queue handler to process events sequentially. A queue handler can be just an array that can store the event or it can be a complex priority queue to ensure that high priority events are processed as soon as possible.



Ready-made FIFO Queue Handling C code routines are included with the visualSTATE software, in the `Examples\SampleCode\` directory.

An event could be handled this way:

- Your main application periodically checks the event queue for newly arrived events. When there is an event to be processed, the proper visualSTATE API functions are called. The traditional way of doing this is to create an infinite loop in your main function that checks for events and calls the API. In this way it is, for example, easy to put the MCU in a low-power mode when there are no new events to process.
- Once the application receives input (which could be the result of an interrupt or manual polling/checking), it is added as an event to the event queue. If the main loop is doing power management, a hardware interrupt or the interrupt routine can wake up the CPU to the appropriate level.

The following two code piece examples are all that must be inserted into your application to process your events.

Example of code for event handling

```
void scanInputs (void)
{
    switch (PIND)
    {
        case BTN1: SEQ_AddEvent( eButton1 ); /* Add EVENT: */
                /* eButton1 to the queue */
                break;
        case BTN2: SEQ_AddEvent( eButton2 ); /* Add EVENT: */
                /* eButton2 to the queue */
                break;
        default:  break;
    }
}
```

Example of visualSTATE API code

```
/* Defines an action expression variable */
SEM_ACTION_EXPRESSION_TYPE actionExpressNo;

/* Defines and initializes. In this case the reset */
/* event is SE_RESET */
SEM_EVENT_TYPE eventNo = SE_RESET;

/* Initializes the VS system. */
SEM_Init();

/* Initializes external variables if used */
/*SEM_InitExternalVariables();*/

/* Initializes internal variables if used */
/*SEM_InitInternalVariables();*/

/* Initializes signal queue if signals are used */
SEM_InitSignalQueue();

if ((cc = DEQ_Initialize()) != UCC_OK)
    HandleError(cc);

/* Do forever */
for(;;)
{
    /* Deducts the event. */
    if ((cc = SEM_Deduct(eventNo)) != SES_OKAY)
        HandleError(cc);

    /* Get resulting action expressions and execute them */
    while ((cc = SEM_GetOutput(&actionExpressNo)) ==
        SES_FOUND)
        SEM_Action(actionExpressNo);
    if (cc != SES_OKAY)
        HandleError(cc);

    /* Changes the next state vector. */
    if ((cc = SEM_NextState()) != SES_OKAY)
        HandleError(cc);

    /* Gets next event from queue */
    while (DEQ_RetrieveEvent(&eventNo) == UCC_QUEUE_EMPTY)
        OS_Wait (OS_VS_EVENT_PENDING, INFINITE);
}
```

Execution

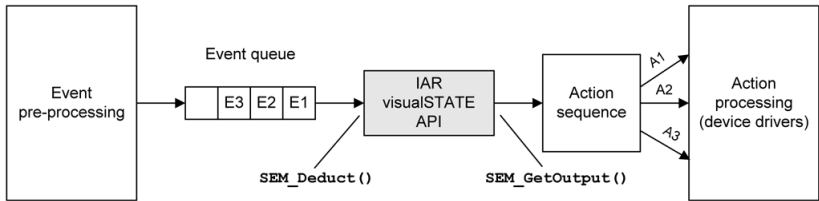
- 1 The next time your `while` loop checks your queue, it will find an event if there is one.
- 2 Upon detecting an event in your queue, you make sure that the `visualSTATE` system is activated (using the `SEM_Deduct` API function) and let `visualSTATE` get back with a list of actions to take (using `SEM_GetOutput`). Use function pointer tables, for example.
- 3 These action functions will be called one at a time.
- 4 Then the API function `SEM_NextState` is called, to change states.
- 5 When the state combination has been updated, the execution goes back to the `while` loop to check your queue for other events to be processed, or perform any other steps required.

Note: When initializing a loop, always supply the reset event as the first event into the queue. The reset event changes the system to the reset state. The default name of the `visualSTATE` reset event is `SE_RESET`.

The API functions used above are from the Table-based API that uses compact binary tables to represent your model plus the API functions mentioned above. You can also generate code based on a traditional `switch/if` structure. The **Readable code generation** is a project-level option on the **Project>Options>Code generation** Configuration page. This format has a slightly different API and it is in most cases enough to call the function `VSDeduct` in the event processing loop.

You can find the API code example in the `Examples\SampleCode` directory. Note that the API functions names in the Basic API can also be prefixed with the system name to make it possible to distinguish between different systems during processing. Use **prefix for API** is a system-level option on the **Project>Options>Code generation** API Functions page.

Placing your events in a queue and retrieving them one by one from the queue for handling, offers many advantages to the user and helps create a well-structured, more organized application:



You can also generate C++ code. This can be an advantage if several instances of a system are needed and you are *not* using RealLink debugging. See the application note `Using_visualSTATE_generated_code_with_Cplusplus.pdf` in the `Examples\SampleCode\CplusplusModel\` directory.

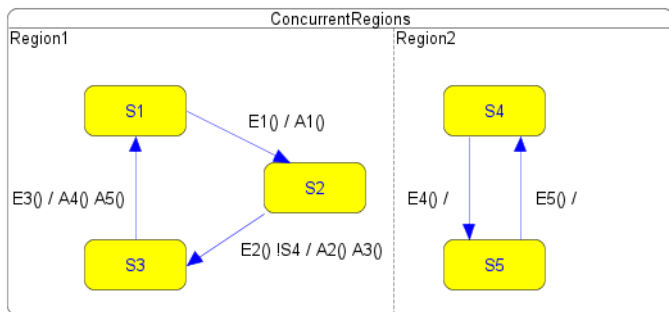
UNDERSTANDING THE VISUALSTATE CONTROL LOGIC CODE

visualSTATE can generate either table-based C code for the state machine logic or a traditionally structured state machine implementation based on `if..else` and `switch/case` statements.

Table-based code generation

In table-based mode, visualSTATE translates your state machine model into table-based C code. This table is then used by the visualSTATE API to process all events, guard expressions, variables, signals, and actions.

This translation is based on the fact that state machines can be expressed in Boolean equations using established mathematical notations, as in this example:



```

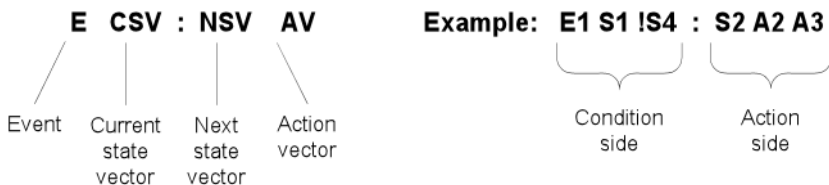
E1 AND S1 -> S2 AND A1
E2 AND S2 AND NOT S4 -> S3 AND A2 AND A3
E3 AND S3 -> S1 AND A4 AND A5
E4 AND S4 -> S5
E5 AND S5 -> S4

```

If we are in state $S1$ and event $E1$ occurs, action $A1$ is executed and the system moves to state $S2$. In other words: IF ($E1 \ \& \ S1$) THEN ($S2 \ \& \ A1$).

Even more complex transitions can be expressed. For instance, if we are in state $S2$ and event $E2$ occurs, and state $S4$ is not active, actions $A2$ and $A3$ are executed and the system moves to $S3$. In other words: IF ($E2 \ \& \ S2 \ \& \ !S4$) THEN ($S3 \ \& \ A2 \ \& \ A3$). This ability to translate state machines from a visual description to logical rules allows us to establish the visualSTATE transition syntax.

This figure shows the transition syntax:



This syntax is used by visualSTATE to generate table-based code using hexadecimal numbers to represent your state machine. Embedded in each

hexadecimal number are the four components of the model logic: the event, the current state vector, the next state vector, and the action vector.

The final result of applying the transition syntax is an ANSI C structure of hexadecimal arrays:

Table of
constant data
representing
the logic of the
visualSTATE
model

```
VSDATA const VS =  
{  
  {  
    0X000, 0X000, 0X001, 0X001, 0X002, 0X002, 0X003, 0X003,  
    0X003, 0X004, 0X004, 0X005, 0X005  
  },  
  {  
    0X000, 0X060, 0X000, 0X002, 0X009, 0X00C, 0X004, 0X008,  
    0X001, 0X003, 0X010, 0X001, 0X000, 0X003, 0X00C, 0X001,  
    0X008, 0X001, 0X010, 0X001, 0X000, 0X001, 0X004, 0X001,  
    0X030, 0X001, 0X001, 0X000, 0X002, 0X009, 0X005, 0X002,  
    0X010, 0X000, 0X001, 0X004, 0X004, 0X002, 0X010, 0X000,  
    0X001, 0X005, 0X005, 0X002, 0X010, 0X000, 0X001, 0X006,  
    0X006, 0X002, 0X010, 0X000, 0X001, 0X007, 0X007, 0X002,  
    0X031, 0X002, 0X003, 0X000, 0X00C, 0X005, 0X002, 0X000,  
    0X009, 0X008, 0X00E, 0X003, 0X031, 0X001, 0X00C, 0X003,  
    0X000, 0X006, 0X00B, 0X003, 0X000, 0X008  
  },  
  {  
    0X00000, 0X000A8, 0X000B2, 0X0003D, 0X00044, 0X0004B,  
    0X00092, 0X0009C, 0X0006B, 0X00075, 0X000E5, 0X000EF,  
    0X00088, 0X000BE, 0X00009, 0X00011, 0X00017, 0X0001F,  
    0X00031, 0X00037, 0X000CB, 0X000D4, 0X000DD, 0X000F9,  
    0X0011B  
  },  
  {  
    0X000, 0X001, 0X003, 0X006, 0X008, 0X00A, 0X00E, 0X012,  
    0X01A, 0X01D, 0X021  
  }  
}
```

The contents of the constant data table are interpreted by the visualSTATE API and turned into executable C code.

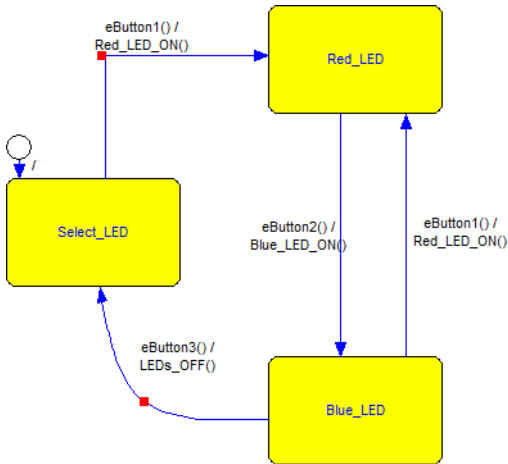
The table represents the entire control logic and allows very tight code generation. The API functions operating on the state machine tables can be regarded as a state machine interpreter. This means that the total code size for a state machine is the sum of the table size plus the size of the used API functions.

Readable code generation

The main purpose of the *readable code* format is to make it easy to map the code back to the design model and make the code easy to understand for a human. The

code generation is based on the same principles as the table generation, which means that the transitions are the basic code generation unit. Consider the example given for table-based code generation above: The transition $E1 \text{ AND } S1 \rightarrow S2 \text{ AND } A1$ can be translated directly to an `if` statement in C. But to make the code easier to read, all transitions triggering on the same event are grouped together in a separate case statement inside a switch statement.

We can take a look at the previously introduced LED state machine model:



We see in the picture that two transitions trigger on the event `eButton1`. The code for these transitions looks like this:

```

case eButton1:
{
    if ((CSV[0] == MY_Topstate_Select_LED))
    {
        Red_LED_ON();
        WSV[0] = MY_Topstate_Red_LED;
    }
    if ((CSV[0] == MY_Topstate_Blue_LED))
    {
        Red_LED_ON();
        WSV[0] = MY_Topstate_Red_LED;
    }
}
break;
  
```

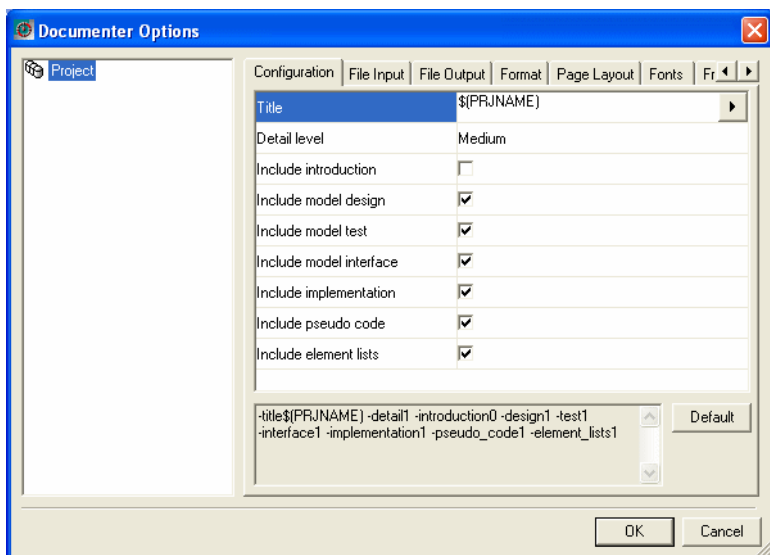
As you see, each transition corresponds to one `if` statement. `WSV` and `CSV` keep track of the current state and the next state. Action function calls, assignments, and guard conditions are generated inline as part of the `if` statement.

Additional features

DOCUMENTING YOUR PROJECT

You can create an up-to-date documentation report of your entire visualSTATE® project using the Documenter.

- 1 In the Navigator, choose **Project>Options >Documentation** to display the **Documenter Options** dialog box.



- 2 On the **Configuration** page, select the section(s) you want to include in your documentation report and the level of detail. Other Documenter options are available on the rest of the pages.

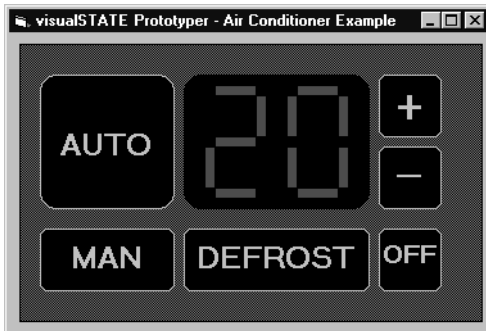
When you are done, click **OK**.

- 3 Choose **Project>Document** to generate the documentation report. When it is finished, a completion message will be written to the Output window.
- 4 The generated documentation report (RTF or HTML) is, by default, located in the `Doc` subdirectory of your project directory. By default, Navigator will try to launch the application associated with the generated file(s), for example Microsoft Word for RTF output.

PROTOTYPING

Because the visualSTATE generated code is target- and CPU-neutral, it is easy to create PC-based GUI prototypes for your application. If you are using C or C++ as your prototyping language, all you need to do is to create a mapping from the GUI framework's event management to visualSTATE events. Action function ports for the GUI-related functionality must also be created.

An example of a prototype for an air conditioner is shown below. In the real hardware version of the air conditioning system, the buttons on the front panel probably either generate interrupts or are polled. The interrupt routines or polling routines can then send the appropriate event to the visualSTATE event queue. But in a prototype, the event handlers for the windowing framework can send events to the visualSTATE model.



If you are familiar with GUI creation in, for example, Visual Basic or Delphi, you can use two different methods:

- Create a DLL from the visualSTATE generated code and interact with that as you would with any other DLL
- Use the prebuilt visualSTATE Expert DLL, a library that you can use with a different visualSTATE model without recompiling it, and that does not require you to compile the visualSTATE generated files. However, to make the DLL work with a model, there are restrictions on the modeling features that can be used.



visualSTATE also offers integration with Altia Design for creating virtual GUI prototypes. With this integration, you can prototype a state machine model and a GUI without writing any code. Altia Design has the added capabilities of automatically generating GUI code for embedded devices. See www.altia.com for more information.