# visualSTATE®

C-SPY®Link User Guide

UVSCL-3

IAR SYSTEMS

## EDITION NOTICE

Third edition: February 2011

Part number: UVSCL-3

This guide applies to version 4.1 and later of IAR Embedded Workbench® and version 6.1 and later of visualSTATE®.

Internal reference: IJOA.

# Contents

# Figures

# Preface

Welcome to the visualSTATE C-SPY®Link User Guide. The purpose of this guide is to provide you with detailed reference information that can help you debug your visualSTATE application in the IAR C-SPY Debugger.

## Who should read this guide

You should read this guide if you want to get the most out of the features in the C-SPYLink interconnection system for debugging visualSTATE applications at state machine level. You need a working knowledge of:

- visualSTATE®
- The IAR C-SPY® Debugger
- The C programming language
- Application development for embedded systems
- The architecture of your microcontroller (refer to the chip manufacturer's documentation)
- The operating system of your host machine.

For a quick introduction to the IAR C-SPY Debugger, see the tutorials available in either the IAR Embedded Workbench® Information Center or in the IAR Embedded Workbench IDE online help system.

## How to use this guide

This guide describes how to debug visualSTATE applications at state machine level using the IAR C-SPY Debugger; this guide does not describe the general features available in C-SPY, visualSTATE, or the hardware target system. To take full advantage of the debugger solution, you must read this guide in combination with:

- Either the *IAR Embedded Workbench IDE User Guide* or the *C-SPY® Debugging Guide*, which describe the general features available in the C-SPY debugger
- The *visualSTATE User Guide* and the *visualSTATE Reference Guide*, which describe the features of visualSTATE
- The documentation supplied with the target board you are using.

Note that additional features may have been added to the software after the *visualSTATE C-SPY®Link User Guide* was created. The release notes supplied with the product contain the latest information.

## Other documentation

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

- Using visualSTATE, refer to the *visualSTATE® User Guide* and the *visualSTATE® Reference Guide*
- Using the IAR Embedded Workbench IDE and the IAR C-SPY Debugger, refer to either the *IAR Embedded Workbench IDE User Guide* or the IAR Embedded Workbench *IDE Project Management and Building Guide* and the *C-SPY® Debugging Guide*, respectively, depending on which IAR Embedded Workbench product you are using
- Programming for IAR compilers, refer to the *IAR C/C++ Compiler Reference Guide* or the *IAR C/C++ Development Guide*, depending on which IAR Embedded Workbench product you are using
- Programming for IAR assemblers, refer to the *IAR Assembler Reference Guide*
- Using the IAR XLINK Linker and library tools, refer to the *IAR Linker and Library Tools Reference Guide*
- Using the IAR ILINK Linker and related tools, refer to the *IAR C/C++ Development Guide*
- Using the IAR DLIB and CLIB libraries, refer to the IAR Embedded Workbench IDE online help system.

All of these guides are delivered in hypertext PDF or HTML format on the installation media.

The IAR Systems web site, **www.iar.com**, holds application notes and other product information.

## Document conventions

This book uses the following typographic conventions:

| **Style** | **Used for** |
|---|---|
| `computer` | Text that you type or that appears on the screen. |

*Table 1: Typographic conventions used in this guide*

| Style | Used for |
|---|---|
| `parameter` | A label representing the actual value you should type as part of a command. |
| **bold** | Names of menus, menu commands, buttons, and dialog boxes that appear on the screen. |
| *reference* | A cross-reference within this guide or to another guide. |
| 🛠 | Identifies instructions specific to the IAR Embedded Workbench IDE interface. |
| ◎ | Identifies instructions specific to the visualSTATE interface. |
| 💡 | Identifies helpful tips and programming hints. |

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

# Debugging visualSTATE applications in C-SPY®

This chapter introduces you to state machine debugging using C-SPYLink, a plug-in solution that makes true high-level state machine debugging of visualSTATE® applications possible in the IAR C-SPY Debugger.

## Product introduction

C-SPYLink bridges visualSTATE 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.

C-SPYLink consists of two parts:

- Extra code and information generation features in visualSTATE
- A plug-in file for C-SPY and the IAR Embedded Workbench IDE.

### FEATURES

C-SPYLink offers the following main features:

- The complete global state of the state machine system can be monitored live
- State machine level breakpoints. Breakpoints can also be set on specific events or signals
- A choice between execution at nearly full speed with periodic updating of the IAR Embedded Workbench IDE windows or at maximum speed without window updates
- No extra user-written support code for communication, configuration of port protocols, etc., is needed.

### REQUIREMENTS

To take full advantage of C-SPYLink, you will need a PC with multi-threading performance. You will also 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. Examples include J-Link® or a general JTAG probe, NEXUS® or hardware emulator support and the corresponding C-SPY driver for the debug system.

## Installation

Support for C-SPY debugging is automatically provided when you install visualSTATE. The support is implemented as an IAR Embedded Workbench plug-in module, `vs.ewplugin`—an XML file that points to a `ValidatorCSpy.dll` file in the visualSTATE installation directory. This file exists in three different versions located in three different subdirectories, to support the different versions of the IAR Embedded Workbench IDE:

- `\visualSTATE 6.`*`n`*`\Bin\ValidatorCSpy.dll` (IAR Embedded Workbench IDE 6.0 and later)
- `\visualSTATE 6.`*`n`*`\Bin\EW5\ValidatorCSpy.dll` (IAR Embedded Workbench IDE 5.x)
- `\visualSTATE 6.`*`n`*`\Bin\EW4\ValidatorCSpy.dll` (IAR Embedded Workbench IDE 4.x).

This DLL file can interact with the debugger to read and write data on the target controller or in the C-SPY Simulator. It can also control the execution of the application on the target hardware or in the simulator.

When visualSTATE is installed it searches for all IAR Embedded Workbench products that are capable of supporting C-SPYLink and installs the plug-in module in the *`cpu_name`*`\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 visualSTATE installation itself. You can copy this file to other locations if needed.

If you install another IAR Embedded Workbench product version after you have installed visualSTATE, you must copy this `vs.ewplugin` file to the IAR Embedded Workbench *`cpu_name`*`\common\plugins` directory of the new product. Make sure that the file path between the `<dllFile>` and `</dllFile>` tags in the `vs.ewplugin` file matches your installation location for visualSTATE. Also make sure that the name of the `ValidatorCSpy` file in the `vs.ewplugin` file reflects your Embedded Workbench version (see above).

If the visualSTATE installation identifies a IAR Embedded Workbench installation with a compatible IDE version, it will also install a build integration plugin file (`swtdvs.dll`) in the *`cpu_name`*`\bin` directory of the Embedded Workbench installation. In addition, a copy of the `swtdvs.dll` file is also available in the `Plugin\` directory of the visualSTATE installation. You can copy this file if you have to install it manually.

### C-SpyLink.c

C-SPYLink needs a small amount of C-SPY-specific code to be included in the IAR Embedded Workbench project.

This support code is collected in a separate source file called `CSpyLink.c`. It is automatically generated in the `Output` directory for the visualSTATE project if **Generate for C-SPYLink** is selected on the **Coder Options Configuration** page in visualSTATE.

`CSpyLink.c` can be present in your IAR Embedded Workbench project at all times, even if the functionality is not used.

## Operating overview

The following figure shows the operating principle behind the C-SPYLink plug-in solution:

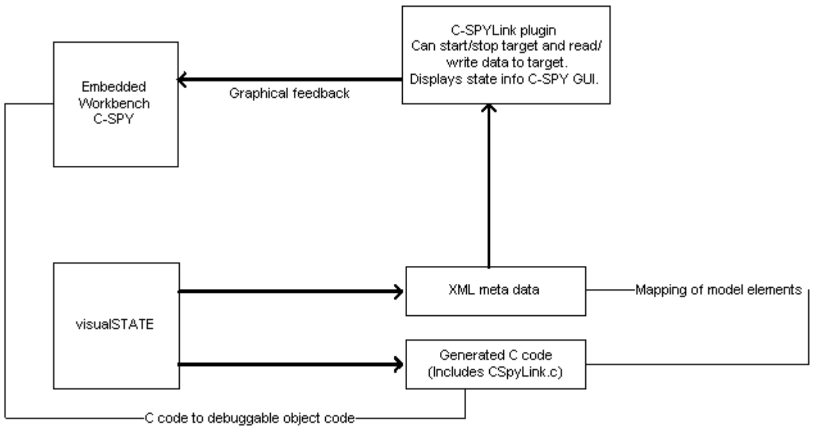

C-SPYLink plugin
Can start/stop target and read/ write data to target.
Displays state info C-SPY GUI.
Embedded Workbench C-SPY
Graphical feedback
visualSTATE
XML meta data
Mapping of model elements
Generated C code (Includes CSpyLink.c)
C code to debuggable object code

*Figure 1: C-SPYLink overview*

Using C-SPYLink in your development project is very straightforward. Configure visualSTATE as described in *Configuration*, page 20, and use the C source files that visualSTATE generates together with the file `CSpyLink.c` in your IAR Embedded Workbench project.

## C-SPYLink execution modes

C-SPYLink can operate in distinct *execution modes*, with different behavior and impact on real-time performance. You set the execution mode by choosing an instrumentation level on the **visualSTATE>Instrumentation Level** submenu (see *The visualSTATE menu*, page 23). The available instrumentation levels are:

- None
- Full instrumentation
- Sampling buffer.

C-SPYLink will set one breakpoint in the target hardware system. The breakpoint serves as a synchronization point for the plug-in module to read data from target memory. This breakpoint will be overloaded with one or more logical breakpoints, which means that execution can stop more than once on the breakpoint for each event processing.

### NONE

If you choose **visualSTATE>Instrumentation Level>None**, your application will execute at full speed, without stops initiated by the plug-in module; the C-SPY windows are not updated until the execution is stopped. Using the None level, only on-target breakpoints can be set and recording an execution sequence can only be performed using the *recording buffer*. (See *On-target breakpoints*, page 16 and *Execution sequences*, page 18.)

If you use the None instrumentation level, execution will be a little slower and the code size will increase. The increase in code size can potentially be significant. The number of allocated breakpoints also affects the execution speed.

**Note:** Full speed can also be achieved by using the Live capture sampling buffer mode, see *Sampling Buffer*, page 15.

### FULL INSTRUMENTATION

In Full instrumentation mode, the IDE windows are continuously updated with detailed information. For each event, you will see which action functions are called and their argument list. This ensures very fine-grained control over what is happening on the target controller at any given point in time.

In this execution mode, the synchronization hardware breakpoint is overloaded with several logical breakpoints. Each time a software breakpoint is hit, data about the system state is read. The logical breakpoints are located so that information in the IDE windows about the system is continuously up to date. The continuous stopping and restarting of execution has a negative impact on runtime performance. This might be a problem in some cases.

The only extra cost in terms of memory, both ROM and RAM, for this mode is the calls to the breakpoint function, which are very inexpensive. The actual overhead depends on the target CPU.

## SAMPLING BUFFER

To enable debugging at close to real-time speeds, you can choose to use the Sampling buffer mode instead.

In Sampling buffer mode, visualSTATE utilizes a buffer on the target hardware to save information about the last complete event processing and the currently ongoing event processing. When execution is stopped, the IDE windows are updated to display the latest information. This information can be accessed during the execution by the C-SPYLink plug-in module if you use either *live capture* or *periodic capture*.

- Live capture

  Using the live capture feature, the plug-in module will read the sampling buffer without stopping the target. Whether this is possible or not depends on the C-SPY debugger driver and it might include adding a small amount of code (see the documentation for your Embedded Workbench product).

  If the debugger driver does not support the feature, warnings are displayed in the Debug Log window, and eventually the feature is disabled.

  Normally there are two sampling buffers, but with live capture the generated code will create a third sampling buffer. This requires some extra RAM.

- Periodic capture

  Using the periodic capture feature, the plug-in module will stop at pre-configured intervals. At each stop, the current completed part of the sampling buffer is read.

The Sampling buffer mode is enabled in the visualSTATE Navigator on a per-system basis. If your visualSTATE project has more than one system that will run in the same application, you can decide per system if you want to have the sampling buffer generated.

The C-SPY commands you choose from the **visualSTATE>Sampling Buffer** submenu in the IAR Embedded Workbench IDE, however, apply to all systems that were generated for using a sampling buffer, if there are more than one.

**Note:** If you use the Sample buffer instrumentation level, execution will be a little slower and the code size will increase. The increase in code size can potentially be significant.

### Full speed with C level breakpoint support

If normal C level breakpoints have been be set in C-SPY, execution will stop whenever a breakpoint is hit. When this happens, the visualSTATE plug-in module will update the affected windows with current system state.

Full speed with C level breakpoint support can be combined with the Sampling buffer mode, to execute at full speed without periodic capture until the breakpoint is hit. When the breakpoint is hit you can then enable periodic capture or the Full instrumentation mode if you wish.

**Note:** Because C-SPYLink always utilizes one breakpoint for synchronization, the number of available breakpoints for C level breakpoints can be an issue if your application executes from ROM or flash memory.

## State machine breakpoints

Using C-SPYLink, you can create a sophisticated data breakpoint called *state machine* breakpoint by specifying a set of goal states from different parallel regions of your state machine system. 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 hit, there are three visual clues that indicate that it was a state machine breakpoint:

- The source window displays a green arrow on the `_VS_breakpoint` function. This is a function used by C-SPYLink as a placeholder for the real C-SPY breakpoint used by visualSTATE to synchronize data, see *C-SPYLink execution modes*, page 13. (This visual clue is not displayed if the breakpoint is a shared DLIB breakpoint or ARM EABI semi-hosting breakpoint, see *Shared DLIB breakpoints*, page 18.)
- The breakpoint number in the Breakpoints window is blinking
- A message that a state machine breakpoint has been hit is displayed in the Debug Log window.

There are two types of state machine breakpoints: *full instrumentation* breakpoints and *on-target* breakpoints. They have the same features, but different performance. A breakpoint can hold information about a trigger (event or signal) and state vector before and after a step.

### FULL INSTRUMENTATION BREAKPOINTS

In Full instrumentation mode, all breakpoints will be treated as full instrumentation breakpoints. They do not take up any extra memory, because the plug-in module handles all checking of breakpoint conditions. There is no limit to the number of full instrumentation breakpoints.

### ON-TARGET BREAKPOINTS

*On-target* breakpoints are in a way similar to the sampling buffer. A *breakpoint buffer* is created in target memory and a small amount of code is generated to check the breakpoint conditions.

To allocate the necessary space in target memory, use the **Number of breakpoints for C-SPYLink** option on the visualSTATE **Coder Options** page; see *Configuration*, page 20. When Full instrumentation mode is disabled, all breakpoints will be regarded as *on-target* breakpoints if there is space allocated for the breakpoint buffer.

On-target 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.

- A *pre-deduct* condition will stop execution *before* processing (deduction of) a new event, but after the complete processing of the preceding event. In effect, this means that it is the result of the previous event processing that will be used as the stop criteria.
- A *post-deduct* condition will stop execution *after* the event processing (deduction) is complete.

A minor difference between these is that a pre-deduct condition is taken when the trigger is injected. The important difference is seen when a 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.

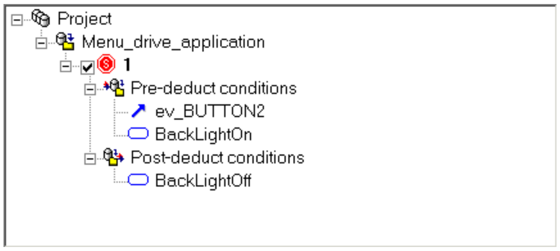


*Figure 2: Breakpoint example*

The breakpoint in Figure 2, *Breakpoint example* is triggered when the BackLightOn state is active. The event ev_BUTTON2 is processed and the resulting state is BackLightOff.

See *Breakpoints window*, page 26, for details about how to set and delete breakpoints.

### SHARED DLIB BREAKPOINTS

C-SPYLink normally allocates a breakpoint that is shared by all visualSTATE debugging features.

If you are using the IAR DLIB runtime environment, you can instead use a shared DLIB breakpoint, to make visualSTATE share the same breakpoint as the C library code for debugging.

To use a shared DLIB breakpoint, follow these steps:

1. Open the visualSTATE Navigator.
2. Choose **Project>Options>Code Generation**.
3. Select the Project in the left-hand pane and open the C-SPYLink page.
4. Select **Enable using shared DLIB breakpoint**.

For IAR Embedded Workbench for ARM 5.1 and later, there is another shared breakpoint that can be enabled in a similar manner. Follow the steps above and select **Enable using ARM EABI shared semi-hosting breakpoint**.

This allows you to save a breakpoint by overloading a visualSTATE breakpoint on a shared debug breakpoint.

## Execution sequences

To help you to debug state machines, you can record an execution sequence of signals, events, actions, changes to variables, and more and save the sequence to an XML file. This XML file can be loaded in the visualSTATE Validator.

Sequences are recorded with one of the following methods:

- Recording buffer
- Sampling buffer
- Full instrumentation.

### RECORDING BUFFER

When you record a sequence using a recording buffer, the execution runs at almost full speed on the target hardware. The target hardware must have enough RAM to record the sequence.

Allocate a buffer in the visualSTATE Navigator by choosing **Project>Options>Code Generation** and set a **Recording buffer size** on the **C-SPYLink** page of the dialog box. See *Configuration*, page 20.

## SAMPLING BUFFER

When you record a sequence using a sampling buffer, the recording is performed by stopping the execution after each macrostep to read out the sampling buffer. This slows execution down considerably more than using the recording buffer, but it requires no extra on-target memory.

## FULL INSTRUMENTATION

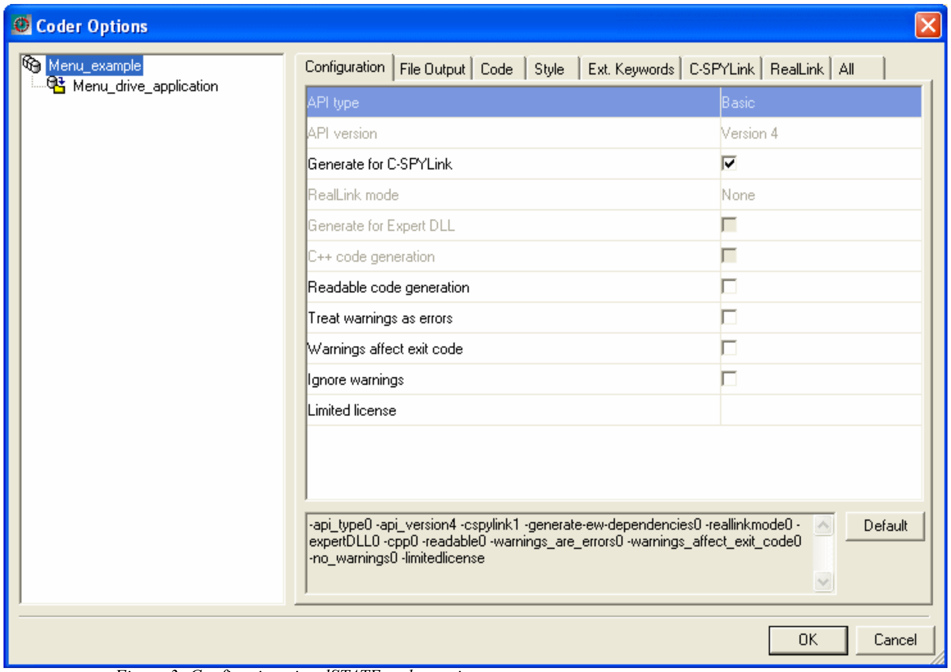
When you record a sequence using full instrumentation, the execution stops frequently. This allows reading out sequences with no extra on-target memory required, but execution is much slower.

To enable recording execution sequences, see *Sequences window*, page 28.

## Configuration

Before you can debug your visualSTATE application in C-SPY, you must enable C-SPYLink in both IAR Embedded Workbench and visualSTATE.

1 In visualSTATE, select the Project in the left-hand pane and choose **Project>Options>Code Generation** to open the **Coder Options Configuration** page.



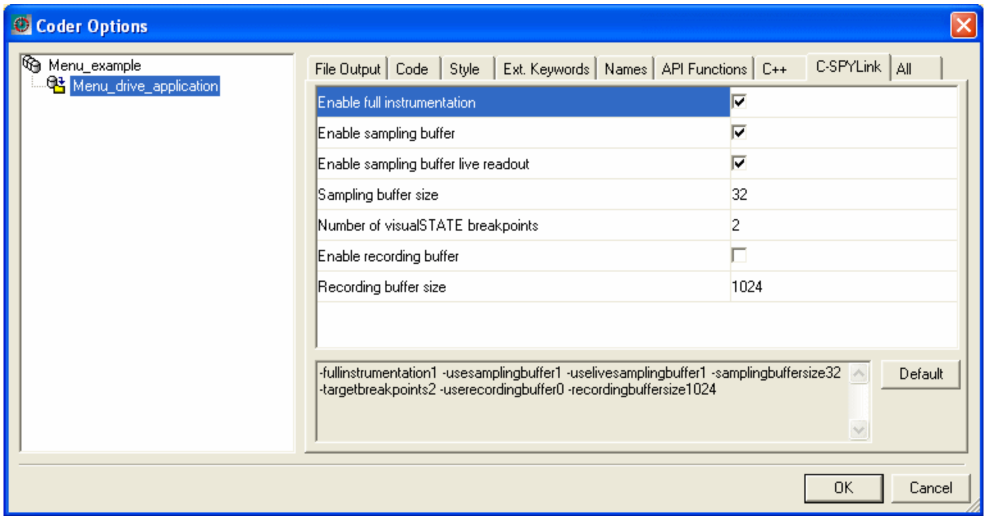
| | |
|---|---|
| API type | Basic |
| API version | Version 4 |
| Generate for C-SPYLink | ☑ |
| RealLink mode | None |
| Generate for Expert DLL | ☐ |
| C++ code generation | ☐ |
| Readable code generation | ☐ |
| Treat warnings as errors | ☐ |
| Warnings affect exit code | ☐ |
| Ignore warnings | ☐ |
| Limited license | |

```
-api_type0 -api_version4 -cspylink1 -generate-ew-dependencies0 -reallinkmode0 -
expertDLL0 -cpp0 -readable0 -warnings_are_errors0 -warnings_affect_exit_code0
-no_warnings0 -limitedlicense
```

*Figure 3: Configuring visualSTATE coder options*

Select the option **Generate for C-SPYLink**.

**2** Select the System you want to debug in the left-hand pane and open the **C-SPYLink** page to set C-SPYLink-specific options.



| | |
|---|---|
| Enable full instrumentation | ✓ |
| Enable sampling buffer | ✓ |
| Enable sampling buffer live readout | ✓ |
| Sampling buffer size | 32 |
| Number of visualSTATE breakpoints | 2 |
| Enable recording buffer | ☐ |
| Recording buffer size | 1024 |

```
-fullinstrumentation1 -usesamplingbuffer1 -uselivesamplingbuffer1 -samplingbuffersize32
-targetbreakpoints2 -userecordingbuffer0 -recordingbuffersize1024
```

*Figure 4: Configuring C-SPYLink options in visualSTATE*

The options must be set to the desired values:

| **Option** | **Description** |
|---|---|
| Enable full instrumentation | See *Full Instrumentation*, page 14. Selected by default. |
| Enable sampling buffer | See *Sampling Buffer*, page 15. Selected by default. |
| Enable sampling buffer live readout | Enables live capture of trace data. See *Sampling Buffer*, page 15. Selected by default. |
| Sampling buffer size | The number of elements (events, signals, states, actions, and transitions) that can be extracted from the sampling buffer. If your design does not use signals, `32` might be enough. If your design uses signals and you want full information from the execution, set this to a higher value. If the value is too small, there will be a message in the Debug log window. By default, this option is set to `32`. |

*Table 2: Description of C-SPYLink coder options*

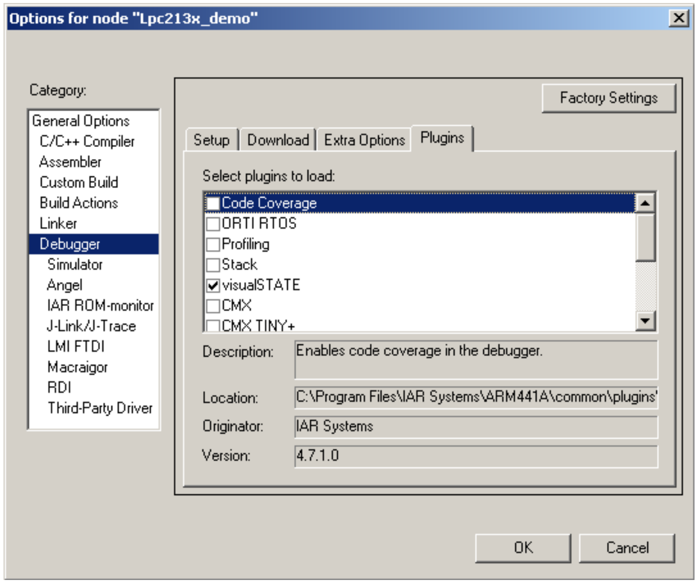
| Option | Description |
|---|---|
| Number of visualSTATE breakpoints | The desired number of breakpoints. When setting it, mind possible hardware and CPU restrictions. The value can be `0`, but if your application is highly dependent on timing or if Full instrumentation mode is very slow, reserving space for breakpoints in the code is better than having C-SPY stop, read, process, and restart to see if any breakpoints were hit.<br>To make breakpoints available also in Sampling buffer mode, the number must exceed `0`. By default, this option is set to `2`. |
| Enable recording buffer | Enables a buffer for recording sequences. See *Execution sequences*, page 18. By default, this option is deselected. |
| Recording buffer size | Sets the size (in elements—events, signals, states, actions, and transitions) of the recording buffer. See *Recording buffer*, page 18. By default, this option is set to `1024`. |

*Table 2: Description of C-SPYLink coder options (Continued)*

Click **OK** to close the dialog box.

**3** After you have generated code in visualSTATE, open your IAR Embedded Workbench project and include the file `CSpyLink.c` in the project, see *C-SpyLink.c*, page 13.

**4** In the IAR Embedded Workbench IDE, choose **Project>Options** and enable C-SPYLink by selecting **visualSTATE** on the **Plugins** page in the **Debugger** category:


Options for node "Lpc213x_demo"

Category:
General Options
C/C++ Compiler
Assembler
Custom Build
Build Actions
Linker
Debugger
Simulator
Angel
IAR ROM-monitor
J-Link/J-Trace
LMI FTDI
Macraigor
RDI
Third-Party Driver

Factory Settings

Setup | Download | Extra Options | Plugins

Select plugins to load:
☐ Code Coverage
☐ ORTI RTOS
☐ Profiling
☐ Stack
☑ visualSTATE
☐ CMX
☐ CMX TINY+

Description: Enables code coverage in the debugger.
Location: C:\Program Files\IAR Systems\ARM441A\common\plugins'
Originator: IAR Systems
Version: 4.7.1.0

OK | Cancel

*Figure 5: Enabling C-SPYLink in IAR Embedded Workbench*

## The visualSTATE menu

When the IAR Embedded Workbench is connected to a visualSTATE project via C-SPYLink, an additional, visualSTATE-specific menu becomes available in C-SPY.

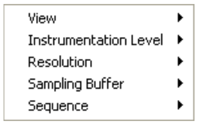


*Figure 6: visualSTATE menu in C-SPY*

The **visualSTATE** menu contains the following commands:

| Menu command | Description |
|---|---|
| View>Actions | Displays the Actions window, where you can view the action functions that are executed as a result of event processing; see *Actions window*, page 25. |

*Table 3: Description of visualSTATE menu commands*

| Menu command | Description |
|---|---|
| View>Breakpoints | Displays the Breakpoints window, where you configure state machine breakpoints; see *Breakpoints window*, page 26. |
| View>Graphical Animation | Displays the Graphical Animation window, with an animation of the execution of the state machine; see *Graphical Animation window*, page 27. |
| View>Sequences | Displays the Sequences window, where you set up the recording of sequences of events, actions, etc; see *Sequences window*, page 28. |
| View>Signal Queues | Displays the signal queues of the application; see *Signal Queues window*, page 29. |
| View>States | Displays the States window which shows the complete system state configuration; see *States window*, page 30. |
| View>Triggers | Displays the Triggers window with all event and signal triggers in the project; see *Triggers window*, page 31. |
| Instrumentation Level | **None**<br>Disables all debugging features except on-target breakpoints and the recording buffer.<br>**Sampling Buffer**<br>Generates data to the sampling buffer; see *Sampling Buffer*, page 15.<br>**Full**<br>Enables Full instrumentation mode, which updates the States window continuously; see *Full Instrumentation*, page 14. |
| Resolution | Specifies what is stored in the sampling and recording buffers in target memory and—if you are using Full instrumentation—what variables that will be read out. If the amount of memory available is too small or if the Full instrumentation execution slows down too much, deselect one or more of the information types. |
| Sampling Buffer | **Live Capture**<br>Enables live capture, which reads the sampling buffer without stopping the target. With live capture, the generated code will create a third sampling buffer. This requires some extra RAM. It is not supported by all C-SPY drivers. See *Sampling Buffer*, page 15.<br>**Periodic Capture**<br>Enables periodic capture, which stops execution at a configurable interval, displays the content of the sampling buffer and then restarts execution again. See *Sampling Buffer*, page 15.<br>**Capture Delay**<br>Sets the minimum interval between each time the debugger stops and reads data from the hardware sampling buffer when periodic capture is used. The default interval is 3 seconds. |

*Table 3: Description of visualSTATE menu commands (Continued)*

| Menu command | Description |
|---|---|
| Sequence | The commands on this submenu operate on the Sequences window, see *Sequences window*, page 28.<br>**Recording Buffer**<br>Enables a sequence buffer in target memory. Select this option to be able to record sequences without stopping execution when you are not using Full instrumentation.<br>**Record All Systems**<br>Creates a new recording sequence in the Sequences window and starts the recording for all systems.<br>**End All Recording**<br>Ends all ongoing recordings of execution sequences in the Sequence window.<br>**Delete All**<br>Deletes all execution sequences in the Sequence window. |

*Table 3: Description of visualSTATE menu commands (Continued)*

## visualSTATE-specific windows in C-SPY

When the IAR Embedded Workbench is connected to a visualSTATE project via the C-SPYLink plug-in solution, a visualSTATE-specific set of windows becomes available from the **visualSTATE** menu in the IAR Embedded Workbench IDE. These are:

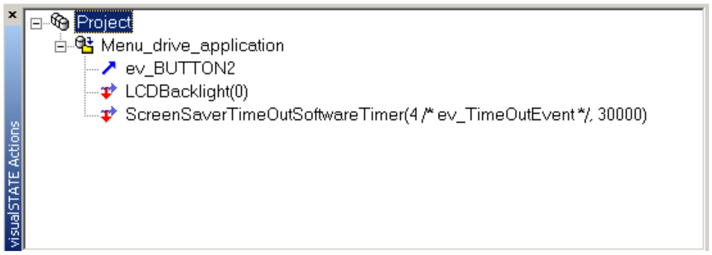
- the *Actions window*
- the *Breakpoints window*
- the *Graphical Animation window*
- the *Sequences window*
- the *Signal Queues window*
- the *States window*
- the *Triggers window*.

### ACTIONS WINDOW

The Actions window—available from the **visualSTATE>View** menu—displays:

- 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).

By right-clicking on a node in the window, you can chose to expand the complete hierarchy.



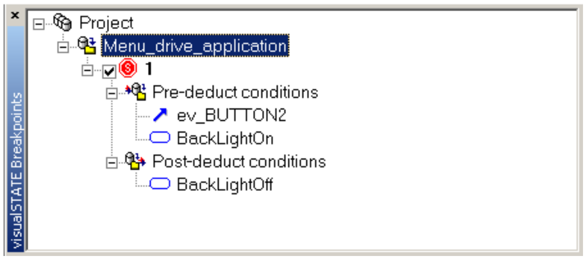
Project
Menu_drive_application
ev_BUTTON2
LCDBacklight(0)
ScreenSaverTimeOutSoftwareTimer(4 /* ev_TimeOutEvent */, 30000)

*Figure 7: Actions window*

When single stepping through the visualSTATE event processing loop in Full instrumentation mode, the window is updated for each completed microstep—that is, every time “something happens”, for example when a transition occurs, an action is performed etc. For information about visualSTATE macrosteps and microsteps, see the *visualSTATE® User Guide*.

## BREAKPOINTS WINDOW

The Breakpoints window—available from the **visualSTATE>View** menu—is used for configuring state machine breakpoints.



Project
Menu_drive_application
1
Pre-deduct conditions
ev_BUTTON2
BackLightOn
Post-deduct conditions
BackLightOff

*Figure 8: Breakpoints window*

To create a state machine breakpoint, right-click on a system and choose **New Breakpoint** from the context menu. States, events, and signals can be dragged from open windows as conditional triggers to the pre-deduct and post-deduct nodes for the breakpoint, see *Pre- and post-deduct conditions*, page 17.

The breakpoint can be enabled and disabled using the check box. Breakpoints can be deleted by right-clicking on the breakpoint node. When the debug session is closed, the breakpoint configuration will be remembered until the next session.

## GRAPHICAL ANIMATION WINDOW

The Graphical Animation window—available from the **visualSTATE>View** menu—shows an animation of the execution of the state machine directly in the original design diagram, as it looks in the visualSTATE Designer. This feature can be active for the specific system or systems you choose.

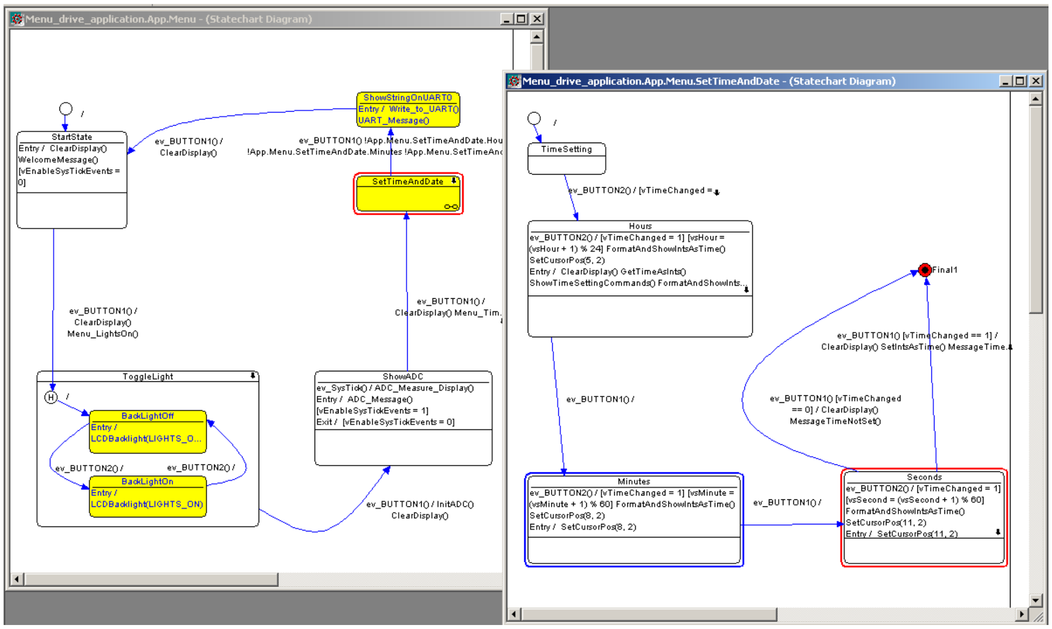

Menu_drive_application.App.Menu - (Statechart Diagram)
StartState
Entry / ClearDisplay() WelcomeMessage() [vEnableSysTickEvents = 0]
ev_BUTTON1() / ClearDisplay()
ShowStringOnUART0
Entry / Write_to_UART0 UART_Message()
ev_BUTTON1() !App.Menu.SetTimeAndDate.Hou
!App.Menu.SetTimeAndDate.Minutes !App.Menu.SetTimeAnd
SetTimeAndDate
ev_BUTTON1() / ClearDisplay() Menu_LightsOn()
ev_BUTTON1() / ClearDisplay() Menu_Tim.
ToggleLight
H
BackLightOff
Entry / LCDBacklight(LIGHTS_O...
ev_BUTTON2() /
ev_BUTTON2() /
BackLightOn
Entry / LCDBacklight(LIGHTS_ON)
ShowADC
ev_SysTick() / ADC_Measure_Display()
Entry / ADC_Message() [vEnableSysTickEvents = 1]
Exit / [vEnableSysTickEvents = 0]
ev_BUTTON1() / InitADC() ClearDisplay()
Menu_drive_application.App.Menu.SetTimeAndDate - (Statechart Diagram)
TimeSetting
ev_BUTTON2() / [vTimeChanged =
Hours
ev_BUTTON2() / [vTimeChanged = 1] [vsHour = (vsHour + 1) % 24] FormatAndShowIntsAsTime() SetCursorPos(5, 2)
Entry / ClearDisplay() GetTimeAsInts() ShowTimeSettingCommands() FormatAndShowInts...
Final1
ev_BUTTON1() [vTimeChanged == 1] / ClearDisplay() SetIntsAsTime() MessageTime.
ev_BUTTON1() /
ev_BUTTON1() [vTimeChanged == 0] / ClearDisplay() MessageTimeNotSet()
Minutes
ev_BUTTON2() / [vTimeChanged = 1] [vsMinute = (vsMinute + 1) % 60] FormatAndShowIntsAsTime() SetCursorPos(8, 2)
Entry / SetCursorPos(8, 2)
ev_BUTTON1() /
Seconds
ev_BUTTON2() / [vTimeChanged = 1] [vsSecond = (vsSecond + 1) % 60] FormatAndShowIntsAsTime() SetCursorPos(11, 2)
Entry / SetCursorPos(11, 2)

*Figure 9: Graphical Animation windows*

Red states indicate newly entered states. Blue states indicate states that were left as the result of the last event processing.

## SEQUENCES WINDOW

The Sequences window—available from the **visualSTATE>View** menu—shows the execution sequences set up for recording, see *Execution sequences*, page 18. To create a new sequence, right-click a system and choose **New Sequence** on the context menu.

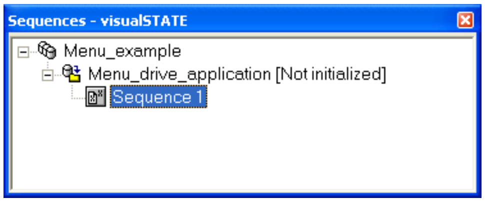


*Figure 10: Sequences window*

To start recording, right-click a sequence and choose **Start Recording** on the context menu. If you are recording using the recording buffer, no data is updated until the buffer in target memory is full or until you stop the recording by right-clicking a sequence and choosing **End Recording** on the context menu. If you are recording using the sampling buffer or using full instrumentation, the sequence data is continuously updated.

If your recording is very long, a node consisting of three periods (. . .) will appear in the middle of a sequence. Right-click it and choose **Expand All** on the context menu to see all nodes. Nodes corresponding to up to 1,000 underlying steps are displayed with the . . . node in the middle—500 steps on each side. Nodes corresponding to a maximum of 100,000 underlying steps can be recorded.

To save a recorded sequence, right-click the sequence and choose **Save** on the context menu. For information about the commands on the **visualSTATE>Sequence** submenu, see *The visualSTATE menu*, page 23.

## SIGNAL QUEUES WINDOW

The Signal Queues window—available from the **visualSTATE>View** menu—shows the status of the signal queue if you are single stepping through the visualSTATE event processing loop.

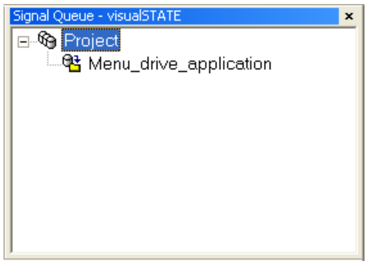


*Figure 11: Signal Queues window*

Normally information about system state is collected after each macrostep, which means that the signal queue is empty by definition. For information about visualSTATE macrosteps and microsteps, see the *visualSTATE® User Guide*.

## STATES WINDOW

The States window—available from the **visualSTATE>View** menu—gives a snapshot of the complete system state configuration. Right-click on a node in the window to expand the complete hierarchy.

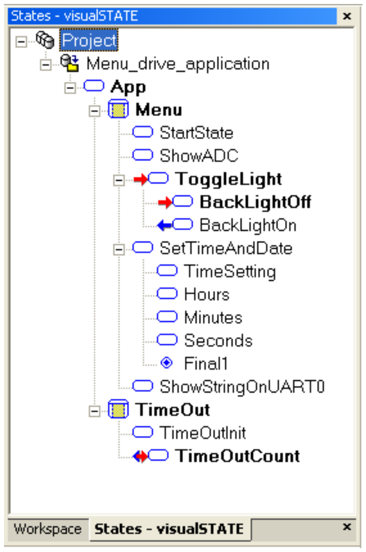


*Figure 12: States window*

Red arrows indicate the states that have become active since the last window update. In Full instrumentation mode this is the same as the states that are the result of the last complete event processing step.

Blue arrows indicate the states that were left as the result of the last complete event processing (macrostep). A blue arrow leaving a state and a red arrow entering the same state indicates either:

- The state has an internal transition or self-transition that triggered in the event processing
- The state is already active and was not deactivated by the last event processing. If Full instrumentation mode is enabled and you are single-stepping through the visualSTATE event processing loop, the States window is updated for each completed microstep.

The difference here between Sampling buffer mode and Full instrumentation mode is that the sampling buffer is only updated occasionally, but when it is updated, the update is complete. In Full instrumentation mode, the window is updated continuously. See *Full Instrumentation*, page 14.

**Note:** The States window is a simplified representation of your state machine design. To see the activity directly in the design model as it looks in the visualSTATE Designer, choose **visualSTATE>View>Graphical Animation**.

### TRIGGERS WINDOW

The Triggers window—available from the **visualSTATE>View** menu—shows all events and signal triggers for the systems that have C-SPYLink enabled.

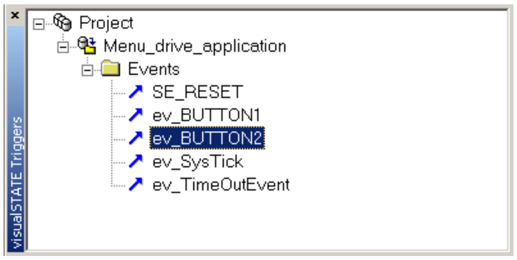


*Figure 13: Triggers window*

Events and signal triggers can be dragged and dropped as event conditions on breakpoints.

## Examples of using C-SPYLink

This section contains examples to help you getting started with C-SPYLink.

### EXECUTION

If your target hardware still has RAM available for a sampling buffer when your application has been compiled with C-SPYLink support, choose **visualSTATE>Instrumentation Level>Sampling Buffer** but deselect **visualSTATE>Sampling Buffer>Periodic Capture**, for an easy-to-follow example of how C-SPYLink works. Then:

- Run your application as usual and try stopping it from time to time or set a breakpoint, to see the current system status as updated in the States window. Running in Sampling buffer mode but without periodic capture results in the smallest impact on real-time performance.
- When the execution is stopped, choose **visualSTATE>Sampling Buffer>Periodic Capture**. Choose **visualSTATE>Sampling Buffer>Capture Delay** to fit your

application's behavior. The default delay is 3 seconds. The restart the execution to see the result.

- Select **visualSTATE>Instrumentation Level>Full**. You will now get continuous updating in the States window. The performance will be slower in this mode.

If your application works by periodically inserting events into the visualSTATE event queue at intervals faster than approximately once per second or once per 0.5 seconds, you might experience a slowdown in your application's response. To change this you can, for example, statically or dynamically lower the frequency of event generation.

## WORKING WITH STATE MACHINE BREAKPOINTS

For an example of how state machine breakpoints work, follow these steps:

1. Enable support for both Full instrumentation mode and Sampling buffer mode on the visualSTATE **Coder Options** page, as described in *Configuration*, page 20. Breakpoints are available in Sampling buffer mode if you set the number of breakpoints to more than 0.
2. Select **Full** or **Sampling Buffer** from the **visualSTATE>Instrumentation Level** submenu.
3. Choose **visualSTATE>View>Breakpoints** to open the Breakpoints window and make sure you have windows open that display the kinds of breakpoint triggers you want to use. In this example the States window will be used.
4. Create a new breakpoint by right-clicking on the system name node in the Breakpoints window:

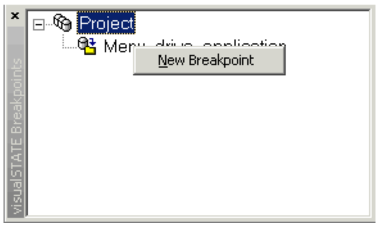


*Figure 14: Setting a new breakpoint*

The new breakpoint will look like this:

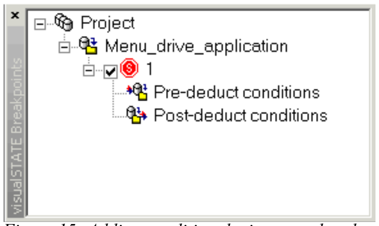
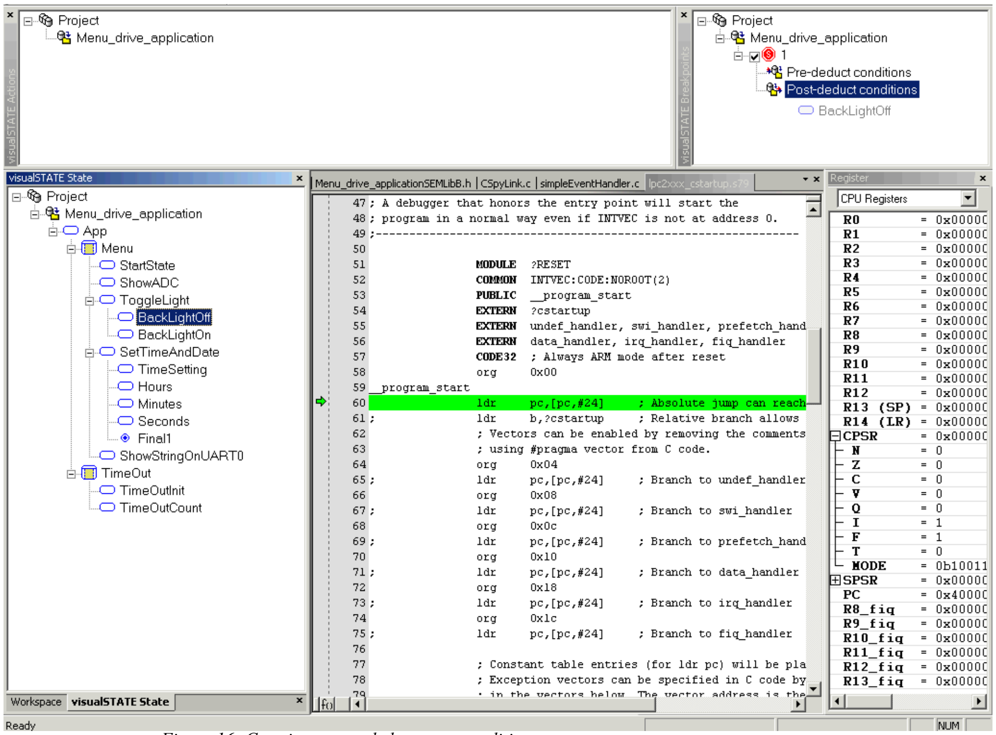


*Figure 15: Adding conditional triggers to breakpoint*

The breakpoint can be enabled and disabled using the check box. It can be set to trigger at two different occasions: before and after an event or signal has been processed. See *Pre- and post-deduct conditions*, page 17.

**5** Add conditional triggers to it by dragging elements from the other windows. For instance, create a post-deduct state condition by dragging one or more states from the States window to the post-deduct node of the breakpoint:



```
; A debugger that honors the entry point will start the
; program in a normal way even if INTVEC is not at address 0.
;-----------------------------------------------------------------

                MODULE  ?RESET
                COMMON  INTVEC:CODE:NOROOT(2)
                PUBLIC  __program_start
                EXTERN  ?cstartup
                EXTERN  undef_handler, swi_handler, prefetch_hand
                EXTERN  data_handler, irq_handler, fiq_handler
                CODE32  ; Always ARM mode after reset
                org     0x00
__program_start
                ldr     pc,[pc,#24]     ; Absolute jump can reach
;               ldr     b,?cstartup     ; Relative branch allows
                ; Vectors can be enabled by removing the comments
                ; using #pragma vector from C code.
                org     0x04
;               ldr     pc,[pc,#24]     ; Branch to undef_handler
                org     0x08
;               ldr     pc,[pc,#24]     ; Branch to swi_handler
                org     0x0c
;               ldr     pc,[pc,#24]     ; Branch to prefetch_hand
                org     0x10
;               ldr     pc,[pc,#24]     ; Branch to data_handler
                org     0x18
;               ldr     pc,[pc,#24]     ; Branch to irq_handler
                org     0x1c
;               ldr     pc,[pc,#24]     ; Branch to fiq_handler

                ; Constant table entries (for ldr pc) will be pla
                ; Exception vectors can be specified in C code by
                ; in the vectors below. The vector address is the
```

```
R0        = 0x00000
R1        = 0x00000
R2        = 0x00000
R3        = 0x00000
R4        = 0x00000
R5        = 0x00000
R6        = 0x00000
R7        = 0x00000
R8        = 0x00000
R9        = 0x00000
R10       = 0x00000
R11       = 0x00000
R12       = 0x00000
R13 (SP)  = 0x00000
R14 (LR)  = 0x00000
CPSR      = 0x00000
 N        = 0
 Z        = 0
 C        = 0
 V        = 0
 Q        = 0
 I        = 1
 F        = 1
 T        = 0
 MODE     = 0b10011
SPSR      = 0x00000
PC        = 0x40000
R8_fiq    = 0x00000
R9_fiq    = 0x00000
R10_fiq   = 0x00000
R11_fiq   = 0x00000
R12_fiq   = 0x00000
R13_fiq   = 0x00000
```

*Figure 16: Creating a post-deduct state condition*

**6** Choose **Debug>Go** to start the execution and watch what happens when the breakpoint is hit.

When you have examined the state of the system, you can restart execution as usual.

## Troubleshooting

This is a short list of issues that might arise:

- 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 to execute in RAM if you need to use breakpoints.
- If the available breakpoints are already used by other C-SPY functionality, the plug-in module will not function properly.

  Here are some examples of breakpoint usage 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, see *Shared DLIB breakpoints*, page 18. If there is no breakpoint available, a workaround is to turn the I/O emulation off on the **Linker** options 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 plug-in modules also need to set a breakpoint. Disable all other plug-in modules and try again.